Treiber schreiben für NetBSD

Jochen Kunz

Version 1.0.1
31. Juli 2003

Eine Einführung in das NetBSD `autoconfig(9)` System und die Prinzipien des Treiberprogrammierens unter NetBSD anhand von Beispielen.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Vorwort

Dieses Dokument soll, allgemeine C Kenntnisse vorausgesetzt, einem Einsteiger die Grundlagen der Unix-Kernelprogrammierung nahe bringen. Als Beispiel dient dabei ein Treiber für ein Floppylaufwerk für NetBSD. Das Floppylaufwerk wurde gewählt da die Hardware und die notwendige Dokumentation dazu - aber eben noch kein Treiber verfügbar war. NetBSD weil es sich auf Grund des klar gegliederten Sourcecodes und der wohldefinierten Schnittstellen hervorragend als Lehrbeispiel eignet.

Leider gibt es kaum spezifische Dokumentation zum Thema Unix-Kernelprogrammierung unter NetBSD, von den Referenzen zu einzelnen Funktionen im 9. Kapitel des NetBSD-Manuals abgesehen. Es fehlt bei diesen Man-Pages aber eine Einführung, ein übergeordnetes Dokument, das die Zusammenhänge verdeutlicht. Dies versuche ich in diesem Dokument zu geben. Ich werde an vielen Stellen auf externe Dokumente, vor allem die des 9. Kapitels des NetBSD-Manuals verweisen. Dieses Dokument soll mehr der „Leim“ zwischen all diesen einzelnen Beschreibungen sein. Im wesentlichen wird dieses Dokument auf den Erfahrungen basieren, die ich durchlitten habe, als ich den Treiber `rf(4)`[1] für den UniBus / QBus RX211 8 Zoll Floppy Controller schrieb.

8 Zoll Floppy? Sowas gab es? Ja. Das waren die ersten Floppies, die Ende der 60’er / Anfang der 70’er Jahre des vergangenen Jahrhunderts gebaut wurden. *UniBus / QBus? Wasndasn?* Das ist der Haus- und Hof-Bus von VAXen[2]. Die VAX war *die* Maschine der späten 70’er bis in die Anfänge der 90’er Jahre des vergangenen Jahrhunderts. Dann wurde sie duch die Alpha Architektur abgelöst. BSD Unix hat eine Lange und ruhmreiche Geschichte auf der VAX. [McK 99] Aber warum schreibe ich heute einen Treiber für derart antike Technik? Im Endeffekt ist es egal ob ich die notwendigen Schritte anhand der neuesten 1GBit/s Ethernetkarte für einen PCIX Bus oder sonst was erkläre. Die Prinzipien sind die selben. Abgesehen davon ist die als Beispiel dienende Hardware recht einfach aufgebaut. Somit wird der Blick auf das Wesentliche frei, anstatt z.B. von PeeCee Idiotismen versperrt zu werden.

Das nun folgende Kapitel gibt einen kurzen Überblick über das `autoconf(9)` Konzept von NetBSD. Einige Details habe ich mir dabei erspart zu beschreiben und verweise stattdessen auf die entsprechenden man-Pages um nicht Informationen zu duplizieren.

Das dritte Kapitel dokumentiert die Implementierung der `autoconf(9)` Schnittstelle von `rf(4)`.

Das vierte und letzte Kapitel behandelt den eigentlichen Treiber, also die

[1] *R*X01/02 *F*loppy
[2] Plural für VAX

Funktionen des Treibers, die Daten von und zu dem physikalischen Gerät transportieren.

Im Anhang findet sich der vollständige Sourcecode des Treibers sowie eine Kopie der referenzierten Man-Pages.

Ausblick: Dieses Dokument ist in seiner jetzigen Form sicher nur ein Anfang. Es bieten sich zukünfige Erweiterungen wie die Beschreibung eines Netzwerkkartentreibers an. Oder eine Beschreibung der internen Funktionsweise von `bus_space(9)` und `bus_dma(9)` oder was notwendig ist um NetBSD auf eine neue Architektur zu portieren. Auch eine Beleuchtung der UVM / UBC Interna oder Dateisystemschnittstellen ist sicher für viele von Interesse um z.B. ein neues Dateisystem zu implementieren. Doch zumindest letzteres entfernt sich zu weit von der ursprünglichen Intension einen Einstieg in das Treiberprogrammieren zu geben und ist eher etwas für ein umfassendes Werk über NetBSD Kernelinterna, das eines Tages vielleicht aus diesem Text erwächst.

Dank an Hubert Feyrer und Marc Balmer, die sich die Zeit nahmen meine geistigen Ergüsse korrektur zu lesen und einige Anregungen für Diagramme gaben.

2 Das autoconf(9) System

Die Kernelkonfiguration basiert auf drei Säulen: config(8), ioconf.c / cfdata und sys/kern/subr_autoconf.c Dieses Konzept ist allgemein unter *autoconf* bekannt geworden. Aber was geht dabei hinter den Kulissen ab?

2.1 config(8)

Bei BSD Unix Kerneln gibt es eine zentrale Datei mit der die Kernelkonfiguration deklariert wird. Bei NetBSD liegt diese in sys/arch/<arch>/conf. <arch> ist dabei die jeweilige Maschinen- / Prozessorarchitektur, in unserem Beispiel also vax, d.h. sys/arch/vax/conf. In diesem Verzeichnis findet sich die Kernelkonfigurationsdatei GENERIC, in welcher alle von dieser Maschine unterstützten Treiber und sonstige Optionen aktiv sind. Man erstellt eine benutzerdefinierte Kernelkonfiguration in dem man diese Datei auf einen neuen Namen im selben Verzeichnis kopiert und editiert. I.d.R. bedeutet editieren dabei einfach alle Treiber für Geräte auszukommentieren, die in der konkret gegebenen Zielhardware nicht vorhanden sind. Dies kann mit dem Tool pkgsrc/sysutils/adjustkernel auch automatisch erfolgen.

Beim Aufruf von config(8) liest es die Kernelkonfigurationsdatei um festzustellen welche Treiber / Funktionen in den Kernel rein sollen. Einige „files.*“ Dateien ordnen dabei den verschiedenen Treibern und Funktionen .c- und .h-Dateien zu und beschreiben Abhängigkeiten zwischen den .c- und .h-Dateien. Anhand dieser Zuordnungen und Abhängigkeiten erstellt config(8) ein Kompilierverzeichnis, das ein Makefile sowie einen Satz .c- und .h-Dateien enthält. Die .h-Dateien enthalten i.d.R. defines mit Parametern wie die max. Anzahl von Treiberinstanzen (z.B. PseudoTTYs, BPF, ...), Kerneloptionen wie KTRACE, ... Die Datei param.c fällt ebenfalls in diese Kategorie.

Das Kompilierverzeichnis trägt den selben Namen wie die Kernelkonfigurationsdatei und liegt in sys/arch/vax/compile. Nach dem Wechsel in das Kompilierverzeichnis wird die eigentliche Kernelcompilation mittels make depend netbsd durchgeführt. Siehe auch config(8) und http://www.netbsd.org/Documentation/kernel/.

2.2 ioconf.c und cfdata

Die Datei ioconf.c im Kompilierverzeichnis enthält die Datenstruktur, die der zentrale Dreh- und Angelpunkt des ganzen *autoconf* Prozesses ist. Diese *configuration data* Tabelle bildet alle von diesem Kernel unterstützten Geräte ab. Fangen wir mit einem Ausschnitt der der Kernelkonfigurationsdatei an:

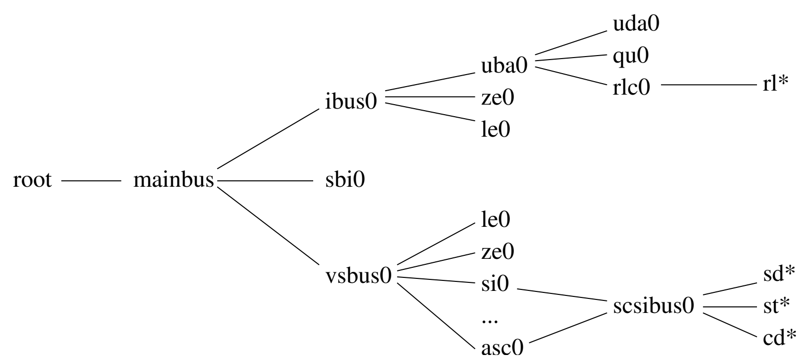

Abbildung 1: Gerätebaum

```
mainbus0        at root

ibus0           at mainbus0             # All MicroVAX
sbi0            at mainbus0             # SBI, master bus on 11/780.
vsbus0          at mainbus0             # All VAXstations

uba0            at ibus0                # Qbus adapter
ze0             at ibus0                # SGEC on-board ethernet
le0             at ibus0                # LANCE ethernet (MV3400)

le0             at vsbus0 csr 0x200e0000 # LANCE ethernet
ze0             at vsbus0 csr 0x20008000 # SGEC ethernet
si0             at vsbus0 csr 0x200c0080 # VS2000/3100 SCSI-ctlr
asc0            at vsbus0 csr 0x200c0080 # VS4000/60 (or VLC) SCSI-ctlr
asc0            at vsbus0 csr 0x26000080 # VS4000/90 SCSI-ctlr

uda0            at uba? csr 0172150     # UDA50/RQDX?
qe0             at uba? csr 0174440     # DEQNA/DELQA
rlc0            at uba? csr 0174400     # RL11/RLV11 controller
rl*             at rlc? drive?          # RL01/RL02 disk drive

scsibus*        at asc?
scsibus*        at si?

sd*             at scsibus? target? lun?
st*             at scsibus? target? lun?
cd*             at scsibus? target? lun?
```

Wir erkennen eine *baumartige Organisation der Gerätetreiber* in Abbildung 1. Am imaginären Übervater `root`, der aus dem „Nichts“ kommt, hängt sein Kind, der abstrakte `mainbus`. Der `mainbus` wiederum ist der Vater der Kinder `ibus`, `sbi` und `vsbus`. Auch diese Kinder sind ihrerseits Väter von `uba`, `le`, `asc`, ... Diese „verwandschaftlichen“ Beziehungen repräsentiert die oben angesprochene `cfdata` Tabelle in der Datei `ioconf.c`. Der Programmierer muß sich nicht um diese Tabelle kümmern. Sie wird von `config(8)` automagisch erzeugt.

Wichtig zu erkennen: Jedes Gerät (Knoten) hat einen Vater (`root` ausgenommen, wegen dem Henne-Ei Problem). Jeder Knoten, der Kinder hat, ist ein Bus oder Controller. Die „eigentlichen“ Geräte sind die Blätter des Baumes. Jedes Blatt und jeder Knoten entsprechen einem Grätetreiber. Das bedeutet also, dass es auch für Bussysteme Treiber geben muss. Diese Bustreiber sind, speziell in diesem Zusammenhang, dafür zuständig die am Bus angeschlossenen Geräte zu suchen/finden (also den „Busscan“).

Zweite wichtige Erkenntniss: Es gibt mehrere, verschiedenne Wege zum selben Treiber! Beispiel `le0`: `root => mainbus0 => ibus0 => le0` oder alternativ `root => mainbus0 => vsbus0 => le0`. `le` ist der eigentliche Treiber für den LANCE Ethernet Chip. Da dieser Treiberkern aber nur über *abstrakte, busunabhängige* Funktionen auf die Hardware zugreift, sind die speziellen Feinheiten der Hardware vor ihm verborgen. Anstatt unmittelbar die Hardware zu manipulieren, hat der Treiber nur abstrakte Handles. Diese Handles und die dazugehörigen Funktionen werden dem Treiber von seinem Vater (also dem Treiber für das Bussystem) bereitgestellt. Freilich müssen alle möglichen Väter (in diesem Falle also `vsbus` und `ibus`) des Kindes (in diesem Falle also `le`) die selbe Schnittstelle „nach unten“ hin haben.[3] Diese Schnittstellen und die Abhängigkeiten zwischen verschiedenen Treibern und anderen Kernelsubsystemen werden über sogenannte *Attribute* beschrieben. Mehr dazu weiter unten.

2.3 sys/kern/subr_autoconf.c

Die Funktionen der Datei `sys/kern/subr_autoconf.c` nudeln die `cfdata` Tabelle in `ioconf.c` beim Booten durch und klappern so nach und nach den ganzen Gerätebaum ab. Dazu muß jeder Treiber eine spezielle Schnittstelle zu diesen Funktionen implementieren.

Der Leser sollte nun (vorzugsweise in dieser Reihenfolge) die Manpages `driver(9)`, `config(9)`, `autoconf(9)` lesen.

[3]Noch lustiger wird es wenn wir auch noch andere Architekturen in die Betrachtung einbeziehen. `le` kann auch an `tc`, `pci`, `zbus`, `vme`, `dio`, `mainbus`, `sbus`, ... hängen.

2.4 Attribute und Locators

Was in autoconf(9) leider nicht so deutlich wird sind die Unterschiede bzw. Zusammenhänge von interface- und plain Attributes und Locators. Ein *plain Attribute* drückt einfach nur aus, dass ein Treiber eine bestimmte Eigenschaft hat. Z.B. dass der Treiber ein Ethernetinterface oder eine serielle Schnitte antreibt. Dadurch ist es möglich, dass mehrere ähnliche Treiber sich mit diesem Attribut assoziieren und so auf gemeinsame Quelltextteile aufbauen. Sobald ein Treiber in den Kernel eingebaut wird, der ein spezielles Attribut benötigt, werden auch die Quelltextteile in den Kernel eingebaut, die dieses Attribut zur Verfügung stellen. ifnet, ether, tty, isadma, ... sind z.B. solche plain Attributes.

Ein *interface Attribute* beschreibt eine logische Softwareschnitstelle zwischen Geräten, typischerweise zwischen Bustreibern und den darunter hängenden Treibern. Es besitzt typischerweise mindestens einen (oder auch mehrere) sogenannte „Locators". Ein Locator gibt die „Position" auf dem Bus / Controller an, an der sich ein Kindgerät befindet. In obiger Kernelkonfigurationsdatei gibt es z.B. das Gerät qe, das am uba[4] hängt, d.h. der QBustreiber implementiert die mit dem Attribut uba gekennzeichnette Softwareschnitstelle auf die sich das Gerät qe bezieht. csr ist dabei der (einzige) Locator des interface Attributs uba.

```
device  uba { csr }
file    dev/qbus/uba.c                        uba

# DEQNA/DELQA Ethernet controller
device  qe: ifnet, ether, arp
attach  qe at uba
file    dev/qbus/if_qe.c                      qe
```

Obiges ist ein Ausschnitt aus sys/dev/qbus/files.uba. Die erste Zeile führt das interface Attribute uba mit dem Locator csr ein. Die folgende Zeile weist config(8) an die Datei dev/qbus/uba.c in die Kernelcompilation einzubinden wenn sich ein Gerät mit dem Attribut uba assoziiert. Die drei letzten Zeilen definieren das Gerät qe. Es ist mit den drei plain Attributes ifnet, ether, arp assoziiert, „hängt" am interface Attribute uba und der zugehörige Quelltext steht in der Datei dev/qbus/if_qe.c.

Ein interface Attribut mit mehreren Locators ist z.B. isa, das die Locators port, size, iomem, iosiz, irq, drq, drq2 unterstützt. Siehe die Deklaration in sys/dev/isa/files.isa. Diese in der Kernelkonfigurationsdatei angegebenen Locators führen direkt zu entsprechenden Werten in dem void *aux

[4] Vor dem QBus gab es den sehr ähnlichen UniBus, *U*ni*B*us *A*dapter wurde dann zu uba. Da die beiden Busse so ähnlich sind reicht ein Bustreiber für beide.

Parameter der foo_match und foo_attach Funktionen. (driver(9) gelesen? ;-))

Locators müssen nicht mit absoluten Werten besetzt werden. Es sind, je nach Vermögen des Treibers, auch Wildcards erlaubt. Ein typischer Kandidat für Wildcards ist ein Bus- oder Controllertreiber, der direct configuration unterstützt. In der „files.*“ Datei, die das interface Attribute mit seinen Locators definiert, müssen in dem Fall für den Locator Standardwerte vergeben werden. Typische Standardwerte sind 0 für Busadressen oder -1 für allgemeine Indizes. Im Kapitel 3.1.2 folgt ein Beispiel für einen solchen Fall. Werden keine Standardwerte vergeben, muß in der Kernelkonfigurationsdatei ein Wert angegeben werden, es sind keine Wildcards erlaubt. Ein in [] deklarierter Locator ist optional.

2.5 Wo sind meine Kinder?

Eine Frage sollte dem Leser jetzt aufgekommen sein: Hmm. Ja. So wird mein Treiber / Gerät gefunden. Aber wie und wo sucht / findet mein Treiber denn jetzt seine Kinder? (Wenn der Treiber einen Bus oder Controller antreibt.) Was hat es mit diesen config_search() und config_found_sm() Funktionen auf sich?

Zwei Fälle sind möglich beim Einbinden der Geräte an einem Bus:

direct configuration Die Busadapterhardware stellt eine volständige Liste aller im Moment physikalisch vorhandenen Kind-Geräte bereit. PCI fällt in diese Kategorie. Der Bustreiber kann durch Auslesen des „PCI Configuration Space“ herausfinden welche PCI Geräte im Augenblick vorhanden sind und dann gezieht nur die Treiber der vorhandene Geräte einbinden.

indirect configuration Der andere Fall trifft auf den QBus oder ISA zu. Bei diesen Bussen kann der Bustreiber nur duch „abklappern“ aller Busadressen das (nicht) Vorhandensein eines Gerätes feststellen.

Liegt ein Fall von indirect configuration vor, so hat der Bustreiber die config_search() Funktionen zu verwenden. config_search() „klappert“ die in der cfdata vorhandenen potenziellen Kind-Gerätetreiber ab, sprich es wird die foo_match() Funktion aller potenziellen Kind-Gerätetreiber aufgerufen. Der Bus- / Controllertreiber ruft also nur einmal config_search() auf um alle Kind-Geräte zu finden. Dabei wird die config_attach() Funktion der Treiber der gefundenen Kind-Geräte *nicht* aufgerufen. Der Bus- / Controllertreiber müßte also nach config_search() selbst die cfdata Tabelle nach seinen eben gefundenen Kindern duchsuchen und config_attach() für diese aufrufen. Mir ist kein Treiber aufgefallen der das macht. Wie in autoconf(9) im Abschnitt über config_search() beschrieben, erledigt man das über den func Funktionsparameter von config_search(). Diese func Funktion wird vom Bus- / Control-

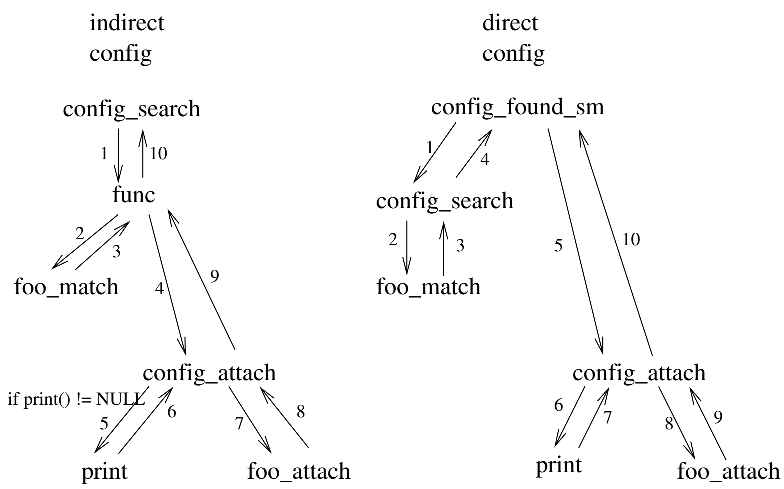

Abbildung 2: Aufrufkette der `autoconf(9)` Funktionen

lertreiber bereitgestellt. `config_search()` ruft für alle in der `cfdata` Tabelle vorhandenen Kind-Gerätetreiber diese `func` Funktion auf. Diese wiederum die `foo_match()` Funktion des Kind-Gerätetreibers und falls diese das Gerät gefunden hat die `config_attach()` Funktion für den Kind-Gerätetreiber.

Unterstützt ein Bus / Controller direct configuration, wie z.B. PCI oder PNP-ISA, ruft der Bustreiber für jedes vorhandene Kind-Gerät die `config_found_sm()` Funktion einmal auf. Diese ruft zuerst `config_search()` und falls das Kind tatsächlich gefunden wurde die `config_attach()` Funktion auf. `config_attach()` reserviert den Speicher für die `softc` Struktur des Kind-Gerätetreibers und ruft dessen `foo_attach()` Funktion auf. Die `submatch()` Funktion ist in diesem Fall oft `NULL`, bzw. wird gleich die `config_found()` Funktion verwendet. Die `print` Funktion stellt der Vater zur Verfügung und gibt der `config_found_sm()` Funktion einen Zeiger zu dieser Funktion als dritten Parameter beim Aufruf mit. Diese `print` Funktion wird innerhalb von `config_attach()` aufgerufen, nach dem `config_attach()` eine Meldung à la „`foo at bar`“ ausgegeben hat. Der Parameter `name` der `print` Funktion ist dabei `NULL`. Die `print` Funktion sollte nähere Informationen zu dem Kindgerät ausgeben auf die Konsole, z.B. genauer Gerätetyp, Übertragungsgeschwindigkeit, ... Ist das nicht gewünscht kann an Stelle des Funktionszeigers auch `NULL` übergeben werden. Und schon hat der Hirte all seine Schafe beisammen. ;-)

Wieso wird mit `config_search()` noch nach dem Kind gesucht, wo der Bus- / Controllertreiber doch schon zweifelsfrei das Vorhandensein des Kind-Gerätes festgestellt hat? Ganz einfach. Das Kind-Gerät mag zweifelsfrei vorhanden sein, ist aber auch sein Treiber im aktuellen Kernel vorhanden? Gibt es für dieses Kind-

Gerät keinen Treiber, schlägt `config_search()` fehl und statt `config_attach()` aufzurufen wird die vom Vater bereitgestellte `print` Funktion direkt aufgerufen. Bei diesem Aufruf allerdings ist der Parameter `name` der `print` Funktion ein Zeiger auf den Namen des Vatergeräts. Die `print` Funktion sollte dann etwas à la „`foo at bar`" ausgeben. Normalerweise würde die `foo_attach` Funktion diese Meldung ausgeben, aber der `foo` Treiber ist ja nicht im Kernel und somit auch nicht seine `foo_attach` Funktion. Die `print` Funktion gibt entweder `UNCONF` oder `UNSUPP` zurück. Dementsprechend wird die Meldung „`not configured`" oder „`unsupported`" an den von der `print` Funktion ausgegebenen Text angehängt. `UNCONF` wird zurückgegeben falls der Treiber im Prinzip existiert, aber nicht in den aktuellen Kernel eincompiliert wurde und `UNSUPP` falls der Vater ein Kind erkannt hat und weiß, dass es noch keinen Treiber dafür gibt. Einen weiteren Grund für die Verwendung der `config_search()` und der `foo_match()` Funktion bei direct configuration erläutert Abschnitt 3.3.3. Doch dieses Detail soll uns erst später verwirren. ;-)

Alles klar? Sicher nicht. Denn wo zum Daemon ruft ein Treiber die `config_found()` oder `config_search()` Funktion denn jetzt auf? Ganz einfach: In seiner `foo_attach()` Funktion. `foo_attach()` initialisiert den Treiber und dazu gehört auch die Suche nach Kindern.

All diese kleinen Puzzlestücke fügen sich zu einem Bild zusammen, wenn man sich mit diesem Wissen bewaffnet in die Niederungen des Kernelquelltextes hinabbegibt und sich die `autoconf(9)` Schnittstelle nebst der involvierten „`files.*`" Dateien einiger bereits vorhandener Treiber ansieht. Ebenfalls interessant ist `sys/kern/subr_autoconf.c`. Besonders die `config_search()` (+`mapply()`), `config_found_sm()` und `config_attach()` Funktionen sind von Interesse (`config_attach()` muß man nicht wirklich detailiert verstehen). Diese wenigen Zeilen Quelltext in `sys/kern/subr_autoconf.c` sind des Pudels Kern.

2.6 `bus_space(9)` und `bus_dma(9)`

Oben habe ich ja bereits angesprochen, dass ein und derselbe Treiberkern an verschiedenen Bussen hängen kann. Mit Bus ist in diesem speziellen Fall ein Systembus à la QBus oder PCI gemeint, dessen Adressbereich in den Adressbereich der CPU eingeblendet wird. Sprich, die CPU kann mit Lade- und Speicheroperationen Daten von und zu Busgeräten transferieren. Nicht in diese Kategorie fallen Massenspeicher-, Desktop- und sonstige Busse wie SCSI, HP-IB, ADB, USB, ... bei denen nur indirekt über einen Hostadapter, womöglich mit einem paketorientierten Protokoll, auf Busgeräte zugegriffen werden kann.

Der Treiberkern selbst benutzt nur abstrakte Funktionen um über Tags und Handles auf die Hardware zuzugreifen. Dadurch wird der Treiberkern unabhängig vom Bussystem. Zu diesem Treiberkern gehören dann u.U. mehrere sogenannte *Bus-Attachments*. Diese Busattachments implementieren die in `driver(9)` beschriebenen `cfattach` und `softc` Datenstrukturen sowie die `foo_match()` und `foo_attach()` Funktionen für jedes Bussystem separat. Die `foo_attach()` Funktion „biegt" dabei die Tags und Handles für den Treiberkern zurecht. `bus_space(9)` und `bus_dma(9)` sind in NetBSD das System um mittels abstrakter Funktionen via Tags und Handles, unabhängig vom benutzten Bussystem, auf die Hardware zuzugreifen. Schreibt man also einen Treiber, erhält man `bus_space(9)` und / oder `bus_dma(9)` Tags und Handles als `aux` Parameter in der `foo_match()` bzw. `foo_attach()` Funktion mitgegeben. Man erhält nicht direkt `bus_space(9)` / `bus_dma(9)` Tags und Handles, sondern einen für den jeweiligen Bus spezifischen Attach-`struct`, der die `bus_space(9)` / `bus_dma(9)` Tags und Handles sowie weitere busspezifische Parameter enthält.

Das `bus_space(9)` / `bus_dma(9)` Tag ist die abstrakte Repräsentation der Bushierarchie einer Maschine insgesamt und das Handle entspricht einer abstrakten Adresse auf einem Bus innerhalb dieser Hierarchie. Der Aufbau dieser Tags und Handles ist stark abhängig von der Architektur der konkreten Hardware. Die Bustreiber für QBus, ISA, ... kümmern sich dabei darum die Locatorwerte aus der Kernelkonfigurationsdatei in entsprechende Handles umzuwandeln. Der eigentliche Treiber muß sich nicht um den Aufbau und Inhalt der Tags und Handles kümmern, sondern sie einfach nur benutzen.

So. Nun wird es aber Zeit `bus_space(9)` und `bus_dma(9)` sowie [Tho] zu lesen. ;-)

3 Der `autoconf(9)` Teil des `rf(4)` Treibers

Und jetzt geht es los. Wir wollen einen Treiber für die VAX schreiben. Der `rf(4)` Treiber wird aus Sicht von `autoconf(9)` aus zwei Treibern bestehen. Einem Treiber für den Controller und einem Treiber für die daran angeschlossenen Laufwerke.

Zum allgemeinen Aufbau des Treiberquelltextes: Der Quelltext sollte mit einen Urheberrechtsvermerk beginnen. Das Copyright des Quelltextes muß „kompatibel“ mit der BSD Lizenz sein. Es bietet sich daher an ebenfalls die BSD Lizenz bzw. eine BSD-artige Lizenz zu verwenden. Als zweites ein Kommentar mit allgemeinen Anmerkungen, wie „Dies ist ein Treiber für bla, ...“, TO-DO Liste, bekannte Bugs / sonstige Unzulänglichkeiten, ... Hiernach die Include-Anweisungen, beginnend mit allgemeinen, kernelweiten Includedateien, zum Ende hin speziellere Includedateien, die nur von diesem Treiber genutzt werden. Als nächstes Päprozessoranweisungen wie Makrodefinitionen und symbolische Konstanten, gefolgt von Funktionsprototypen und der Deklaration von Datenstrukturen und -typen. Siehe dazu auch `/usr/share/misc/style` (Ebenfalls im Anhang enthalten). Dieser Beispielquelltext erläutert die Einrückungsregeln, die für NetBSD-Quelltexte eingehalten werden *müssen*. (Zumindest wenn man den Quelltext irgendwann im NetBSD CVS-Repository sehen will.)

3.1 Konfigurationsdateien

3.1.1 Kernelkonfigurationsdatei

```
rfc0            at uba? csr 0177170     # RX01/02 controller
rf*             at rfc? drive?          # RX01/RX02 floppy disk drive
```

Das ist alles was zur Kernelkonfigurationsdatei hinzugefügt werden muß um den Treiber zu aktivieren. Das `csr` ist der Locator des UniBus / QBus. Er wird im Octalcode angeben und entspricht der Adresse des Gerätes unter der es sich auf dem QBus meldet.

3.1.2 `sys/dev/qbus/files.uba`

Dann noch in `sys/dev/qbus/files.uba`[5] unseren Treiber und die ihn implementierenden Dateien eintragen:

```
# RX01/02 floppy disk controller
device  rfc { drive=-1 }
```

[5] `files.uba` an statt `files.qbus` ist historisch bedingt. Den UniBus gab es vor dem QBus.

```
attach  rfc at uba
device  rf: disk
attach  rf at rfc
file    dev/qbus/rf.c                rf | rfc needs-flag
```

Die erste Zeile macht `config(8)` bekannt, das es einen Treiber `rfc` gibt. Da andere Geräte „unterhalb“ von `rfc` hängen können, `rfc` somit auch ein interface Attribut darstellt, geben wir `config(8)` den Locator `drive` an. Da der `rfc` Treiber direct configuration unterstützt ist es am sinnvollsten Wildcards zu erlauben, also geben wir dem Locator den Standardwert -1. (Siehe 2.4 wieso -1.)

Die zweite Zeile sagt `config(8)`, das `rfc` an dem Bus `uba` (=interface Attribut) hängt. Analog beschreibt die vierte den Zusammenhang zwischen `rf` und `rfc`.

Die dritte Zeile ist dann schon wieder etwas interessanter. Diese Zeile definiert den Treiber `rf` und assoziiert ihn mit dem Attribut `disk`[6].

Die letzte Zeile schließlich sorgt dafür, dass die Datei `sys/dev/qbus/rf.c` mit in den Kernel komiliert wird, falls eines der beiden Attribute `rfc` und / oder `rf` in der aktuellen Kernelkonfiguration auftauchen. `needs-flag` sorgt dafür, dass von `config(8)` im Kompilierverzeichnis eine Datei `rf.h` erzeugt wird. Diese enhält ein `#define NRF 1` und `#define NRFC 1` falls der `rf(4)` Treiber in den Kernel eingebaut wird, und `#define NRF 0` und `#define NRFC 0` falls nicht. Diese Präprozessorkonstanten können von Quelltextbestandteilen benutzt werden, die in irgend einer Weise vom (Nicht-) Vorhandensein des Treibers abhängen.[7]

3.1.3 Die Devicenumbers

Wie sicherlich allen Lesen bekannt bilden Devicenodes, die als Dateien im Dateisystem erscheinen, die Brücke für Userlandprozesse zur Hardware, zu den Treibern im Kernel. Der Kernel unterscheidet die Devicenodes an Hand der Major- und der Minordevicenumber. Jeder Treiber, der Devicenodes unterstützt, hat eine individuelle Majordevicenumber über die er identifiziert wird.

Irgendwo muß der Kernel ja eine Tabelle haben, die ihm sagt welcher Treiber für welche Majordevicenumber zuständig ist. (Die Minordevicenumber handhabt der Treiber eigenständig.) Es gibt für Character- und Blockdevices unabhängige Tabellen, somit kann die Majordevicenumber des Characterdevices eine andere sein, als die des Blockdevices. Seit NetBSD 2.0-current werden diese Ta-

[6] Genau genommen ist `disk` eine „*device class*“. Was auch immer der diffizile Unterschied zwischen einem Attribut und einer „device class“ ist.

[7] Bis inkl. NetBSD 1.6-release sind diese Präprozessorkonstanten auch notwendig für die Handhabe der Majordevicenumbers der Character- und Blockdevicenodes. Daher ist `needs-flag` in diesen Verionen unbedingt notwendig.

bellen automatisch von config(8) in der Datei devsw.c im Kernelkompilationsverzeichnis erstellt.[8] Als Vorlage dient dabei die Liste der Majordevicenumbers in sys/arch/<arch>/conf/majors.<arch>. Dabei ist <arch> die jeweilige Maschinen- / Prozessorarchitektur, in diesem Falle also vax. Somit ist folgende Zeile in sys/arch/vax/conf/majors.vax notwendig um die Majordevicenumbers zu erhalten:

```
device-major   rf              char 78  block 27       rf
```

Die Nummern 78 und 27 ergeben sich einfach daraus, dass der letzte schon vorhandene Eintrag die Nummern 77 und 26 nutzt.

In der devsw.c Datei sind die beiden Tabellen bdevsw und cdevsw[9]. Der Index dieser Tabellen ist die Majordevicenumber. Jede Zeile dieser Tabellen entspricht einem Gerät und enthält einen Zeiger auf die im Treiber deklarierte struct bdevsw bzw. struct cdevsw Datenstruktur, die je nach Gerätetyp mehrere Funktionszeiger enthält:

```
const struct bdevsw rf_bdevsw = {
        rfopen,
        rfclose,
        rfstrategy,
        rfioctl,
        rfdump,
        rfsize,
        D_DISK
};

const struct cdevsw rf_cdevsw = {
        rfopen,
        rfclose,
        rfread,
        rfwrite,
        rfioctl,
        nostop,
        notty,
        nopoll,
        nommap,
        nokqfilter,
        D_DISK
```

[8] Bis inkl. NetBSD 1.6-release finden sich diese Tabellen in sys/arch/<arch>/<arch>/conf.c und müssen von Hand gepflegt werden.

[9] BlockDEViceSWitch respektive CharacterDEViceSWitch. Diese Tabellen gab es übrigens schon in ähnlicher Form und mit gleichen Namen in UNIX V6 von 1976. Siehe [Li 77]

```
};
```

Diese Funktionen implementieren die verschiedenen, mit dem Gerät möglichen Operationen wie `open, close, write, ioctl...`. Implementiert ein Treiber eine der Funktionen nicht, wird an deren Stelle `no<Funktionsname>` geschrieben, z.B. `nommap`. Das letzte Feld in der `cdevsw` bzw. `bdevsw` Datenstruktur ist der Gerätetyp. Z.Z. gibt es die Typen: `D_DISK, D_TAPE, D_TTY`. Blockdevices sind verständlicherweise immer vom Typ `D_DISK`. Der Gerätetyp bestimmt welche Funktionen der Treiber implementieren muß:

D_DISK: `open, close, read, write, ioctl`

D_TAPE: `open, close, read, write, ioctl`

D_TTY: `open, close, read, write, ioctl, stop, tty, poll`

`D_DISK` und `D_TAPE` Geräte müssen also `stop, tty, poll` nicht bereit stellen. (Frage an die Gurus: Was ist mit Treibern, die auch mmap(2) implementieren?) Wie unten zu sehen ist gibt es Präprozessormakros um die Deklaration dieser Funktionen zu vereinfachen und vereinheitlichen. All diese Makros und vorgefertigten Datenstrukturen finden sich übrigens in `sys/sys/conf.h`.

```
dev_type_open(rfopen);
dev_type_close(rfclose);
dev_type_read(rfread);
dev_type_write(rfwrite);
dev_type_ioctl(rfioctl);
dev_type_strategy(rfstrategy);
dev_type_dump(rfdump);
dev_type_size(rfsize);
```

3.2 Datenstrukturen für autoconf(9)

Wie `driver(9)` schon andeutet benutzt der Kernel einen statischen `struct`, über den die notwendigen Funktionen für `autoconf(9)` eingebunden werden. Seit NetBSD 2.0-current erfolgt die Deklaration dieses `structs` mittels eines Makros `CFATTACH_DECL`. Da der `rf(4)` Treiber die „detach“ und „activate“ Funktionen nicht benötigt gibt man an deren Stelle einen `NULL`-Pointer an.

```
CFATTACH_DECL(
        rfc,
        sizeof(struct rfc_softc),
        rfc_match,
```

```
        rfc_attach,
        NULL,
        NULL
);

CFATTACH_DECL(
        rf,
        sizeof(struct rf_softc),
        rf_match,
        rf_attach,
        NULL,
        NULL
);
```

Zuletzt noch den `struct`, mit dem der `rfc` „Vater“ seinem `rf` Kind beim `autoconfig(9)` seine tatsächlich vorgefundene „Busadresse“ mitteilt. Mehr dazu im Abschnitt 3.3.3 über `rf_match` unten.

```
struct rfc_attach_args {
        u_int8_t type;          /* controller type, 1 or 2 */
        u_int8_t dnum;          /* drive number, 0 or 1 */
};
```

3.3 Funktionen für `autoconf(9)`

3.3.1 `rfc_match()`

Ein Vatergerät gibt in dem `void *aux` Parameter eine bus- bzw. controllerspezifische `attach_args` Datenstruktur an seine Kindgeräte weiter. Diese Daten informieren den Kindgerätetreiber „wo“ auf dem Bus nach einem Gerät gesucht werden soll. Bei Erweiterungsbussen wie dem QBus enthält diese Datenstruktur vor allem auch die `bus_space(9)` Handles. Der Treiber kann nur über diese Handles mittels der `bus_space(9)` Funktionen / Makros auf die Hardware zugreifen. Daher ist eine der ersten Aktionen in einer „match“ oder „attach“ Routine ein Typecast des `void *aux` Parameters auf die entsprechende `attach_args` Datenstruktur.

Um mit dem Controller kommunizieren zu können blendet dieser zwei zwei Byte breite „Register“ in den Adressbereich des Busses ein, die sogenannten Command and Status Registers.[10] Wie der Name sagt kann durch Schreiben bestimmter Bitkombinationen via `bus_space_write_2(9)` ein Befehl an den Controller

[10]Wir erinnern uns an den Namen des Locators des UniBus / QBus aus der Kernelkonfigurationsdatei? Er ist „csr“.

gesendet oder sein Status via `bus_space_read_2(9)` abgefragt werden. Je nach Befehl sind mehrere Schreibvorgänge nötig um alle Parameter wie Sektornummer, ... zu übergeben.

```
int
rfc_match(struct device *parent, struct cfdata *match, void *aux)
{
        struct uba_attach_args *ua = aux;
        int i;

        /* Issue reset command. */
        bus_space_write_2(ua->ua_iot, ua->ua_ioh, RX2CS, RX2CS_INIT);
        /* Wait for the controller to become ready, that is when
         * RX2CS_DONE, RX2ES_RDY and RX2ES_ID are set. */
        for (i = 0 ; i < 20 ; i++) {
                if ((bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2CS)
                    & RX2CS_DONE) != 0
                    && (bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2ES)
                    & (RX2ES_RDY | RX2ES_ID)) != 0)
                        break;
                DELAY(100000);  /* wait 100ms */
        }
        /*
         * Give up if the timeout has elapsed
         * and the controller is not ready.
         */
        if (i >= 20)
                return(0);
        /*
         * Issue a Read Status command with interrupt enabled.
         * The uba(4) driver wants to catch the interrupt to get the
         * interrupt vector and level of the device
         */
        bus_space_write_2(ua->ua_iot, ua->ua_ioh, RX2CS,
            RX2CS_RSTAT | RX2CS_IE);
        /*
         * Wait for command to finish, ignore errors and
         * abort if the controller does not respond within the timeout
         */
        for (i = 0 ; i < 20 ; i++) {
                if ((bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2CS)
```

```
                    & (RX2CS_DONE | RX2CS_IE)) != 0
                    && (bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2ES)
                    & RX2ES_RDY) != 0 )
                        return(1);
                DELAY(100000);  /* wait 100ms */
        }
        return(0);
}
```

Zuerst wird dem Controller mal ein Reset übergebraten und max. zwei Sekunden darauf gewartet, dass er das Kommando bestätigt. Falls der Controller den Reset ordnungsgemäß beendet, wird ein weiteres Kommando mit freigeschaltetem Interruptenablebit an ihn gesendet. Warum mit Interrupt, wo doch `driver(9)` sagt, dass das ganze `autoconfig(9)` Prozedere abläuft, wenn Interrupts noch nicht freigeschaltet sind? Nun, das besagt nur, dass ein Treiber noch keine Interrupts nutzen kann, da die ganze Interruptverwaltung des Kernels noch nicht fertig initialisiert ist. Ein Gerät kann dennoch einen Interrupt auslösen, er bleibt nur in den Tiefen der Hard-/Software stecken. Der Interrupt geht verloren, der Interrupthandler wird nicht aufgerufen. Mehr zu Interrupts später.

In diesem Fall ist das sogar zwingend notwendig. Ein Treiber für ein QBus Gerät *muß* in seiner `foo_match()` Funktion einen Interrupt auslösen. Dieser wird vom QBus Bustreiber abgefangen. Das ist die einzige Möglichkeit für den QBus Bustreiber Interruptlevel und -vektor des Geräts zu ermitteln. Folglich gibt der QBus Bustreiber auch eine Fehlermeldung beim Booten aus, falls eine `foo_match()` Funktion das Vorhandensein eines Geräts anzeigt, aber kein Interrupt auftrat. (Das alles macht die Funktion `sys/dev/qbus/uba.c:ubasearch()`, sie ist der `func` Funktionsparameter von `config_search` im QBus Bustreiber. Siehe 2.5)

3.3.2 `rfc_attach()`

Die `attach_args` Datenstruktur ist nur temporär zum `autoconfig(9)` Zeitpunkt gültig. Also kopiert der Treiber später noch benötigte Informationen aus der `attach_args` Datenstruktur in seine `softc` Datenstruktur (mehr zu dieser im nächsten Kapitel) und initialisiert einige erst später benutzte Variablen. Dazu gehört es auch entsprechende Ressourcen wie die „DMA Map" zu reservieren und den Interrupthandler anzumelden. Was und wie dies alles zu tun ist, hängt natürlich sehr stark von dem anzutreibenden Gerät und dem Bus-/Controller an dem es hängt ab.

```
void
rfc_attach(struct device *parent, struct device *self, void *aux)
```

```
{
        struct rfc_softc *rfc_sc = (struct rfc_softc *)self;
        struct uba_attach_args *ua = aux;
        struct rfc_attach_args rfc_aa;
        int i;

        rfc_sc->sc_iot = ua->ua_iot;
        rfc_sc->sc_ioh = ua->ua_ioh;
        rfc_sc->sc_dmat = ua->ua_dmat;
        rfc_sc->sc_curbuf = NULL;
        /* Tell the QBus busdriver about our interrupt handler. */
        uba_intr_establish(ua->ua_icookie, ua->ua_cvec, rfc_intr, rfc_sc,
            &rfc_sc->sc_intr_count);
        /* Attach to the interrupt counter, see evcnt(9) */
        evcnt_attach_dynamic(&rfc_sc->sc_intr_count, EVCNT_TYPE_INTR,
            ua->ua_evcnt, rfc_sc->sc_dev.dv_xname, "intr");
        /* get a bus_dma(9) handle */
        i = bus_dmamap_create(rfc_sc->sc_dmat, RX2_BYTE_DD, 1, RX2_BYTE_DD, 0,
            BUS_DMA_ALLOCNOW, &rfc_sc->sc_dmam);
        if (i != 0) {
                printf("rfc_attach: Error creating bus dma map: %d\n", i);
                return;
        }
```

Dem Gerät zur Initialisierung noch einen Reset überbraten ist an dieser Stelle eine „Gute Idee" (C) (R) (TM), denn eine „attach" Routine darf sich nicht auf irgendwelche „Vorleistungen" der „match" Routine verlassen.

```
        /* Issue reset command. */
        bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS, RX2CS_INIT);
        /*
         * Wait for the controller to become ready, that is when
         * RX2CS_DONE, RX2ES_RDY and RX2ES_ID are set.
         */
        for (i = 0 ; i < 20 ; i++) {
                if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
                    & RX2CS_DONE) != 0
                    && (bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2ES)
                    & (RX2ES_RDY | RX2ES_ID)) != 0)
                        break;
                DELAY(100000);  /* wait 100ms */
        }
```

```
        /*
         * Give up if the timeout has elapsed
         * and the controller is not ready.
         */
        if (i >= 20) {
                printf(": did not respond to INIT CMD\n");
                return;
        }
```

OK. Der Controller ist gefunden. Nach dem unsere `rfc_match()` Funktion das Vorhandensein eines passenden Geräts gemeldet hat gibt der QBus Treiber etwas à la
`rfc0 at uba0 csr 177170 vec 264 ipl 17`
ohne `\n` aus. So kann unsere `rfc_attach()` Funktion noch nähere Informationen zu dem Gerät ausgeben. Und genau das machen wir auch als erstes: Feststellen ob es ein RX01 oder ein RX02 ist, diese Information in der `softc` Struktur speichern und einen entsprechenden Text ausgeben.

```
        /* Is ths a RX01 or a RX02? */
        if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
            & RX2CS_RX02) != 0) {
                rfc_sc->type = 2;
                rfc_aa.type = 2;
        } else {
                rfc_sc->type = 1;
                rfc_aa.type = 1;
        }
        printf(": RX0%d\n", rfc_sc->type);
```

Die letzte Aufgabe ist dann nach den Kindern zu suchen, also festzustellen ob und wo und wie Floppylaufwerke angeschlossen sind. Diese werden sodann mittels `config_found()` in den Gerätebaum aufgenommen.

```
#ifndef RX02_PROBE
        /*
         * Both disk drives and the controller are one physical unit.
         * If we found the controller, there will be both disk drievs.
         * So attach them.
         */
        rfc_aa.dnum = 0;
        rfc_sc->sc_childs[0] = config_found(&rfc_sc->sc_dev, &rfc_aa,rf_print);
        rfc_aa.dnum = 1;
```

```
        rfc_sc->sc_childs[1] = config_found(&rfc_sc->sc_dev, &rfc_aa,rf_print);
#else /* RX02_PROBE */
        /*
         * There are clones of the DEC RX system with standard shugart
         * interface. In this case we can not be sure that there are
         * both disk drives. So we want to do a detection of attached
         * drives. This is done by reading a sector from disk. This means
         * that there must be a formated disk in the drive at boot time.
         * This is bad, but I did not find an other way to detect the
         * (non)existence of a floppy drive.
         */
        if (rfcprobedens(rfc_sc, 0) >= 0) {
                rfc_aa.dnum = 0;
                rfc_sc->sc_childs[0] = config_found(&rfc_sc->sc_dev, &rfc_aa,
                    rf_print);
        } else
                rfc_sc->sc_childs[0] = NULL;
        if (rfcprobedens(rfc_sc, 1) >= 0) {
                rfc_aa.dnum = 1;
                rfc_sc->sc_childs[1] = config_found(&rfc_sc->sc_dev, &rfc_aa,
                    rf_print);
        } else
                rfc_sc->sc_childs[1] = NULL;
#endif /* RX02_PROBE */
        return;
}
```

3.3.3 rf_match()

Der `rfc` Treiber unterstützt direct configuration, ruft also nur dann `config_found()` (und darüber auch `rf_match()`) auf, wenn das Gerät auch zweifelsfrei vorhanden ist. Also könnte man denken kein Problem, das Gerät ist ja da, also reduziert sich die Funktion `rf_match` auf ein simples `return( 1);`. Zu früh gefreut.

```
int
rf_match(struct device *parent, struct cfdata *match, void *aux) {
    struct rfc_attach_args *rfc_aa = aux;

    if ( match->cf_loc[RFCCF_DRIVE] == RFCCF_DRIVE_DEFAULT ||
        match->cf_loc[RFCCF_DRIVE] == rfc_aa->dnum ) {
```

```
        return( 1);
    }
    return( 0);
}
```

Wieso diese Überprüfung? Oder besser *was* wird da überprüft? Die `match` Datenstruktur vom Typ `cfdata` beschreibt die `autoconfig(9)` Parameter dieses Treibers aus der Kernelkonfigurationsdatei . Das Array `cf_loc` der `cfdata` Datenstruktur enthält die Werte der Locators. Die Position der Locators in diesem Array vergibt `config(8)`. Um auf den Wert eines bestimmten Locators in diesem Array zugreifen zu können gibt es Präprozessorkonstanten die der Namenskonvention folgen: `<ATTR>CF_<LOC>` Dabei ist `<ATTR>` der Name des interface Attribute zu dem der Locator gehört, und `<LOC>` ist der gewünschte Locator. Der `rf` Treiber „hängt“ an dem interface Attribute `rfc` mit dem Locator `drive`, also `RFCCF_DRIVE`. So kann der Treiber direkt auf den in der Kernelkonfigurationsdatei angegebenen Wert des Locators `drive` zugreiben. Steht dort z.B.:

```
rf0 at rfc0 drive 1
```

so ist der Wert von `match->cf_loc[RFCCF_DRIVE]` gleich 1. Wird dem Locator in der Kernelkonfigurationsdatei kein Wert zugewiesen, sondern ein Wildcard verwendet, so ist der Wert gleich dem in der Datei `files.uba` angegebenen Standardwert. Dieser Standardwert ist als Präprozessorkonstante `RFCCF_DRIVE_DEFAULT` verfügbar und in unserem Beispiel -1. (Siehe 3.1.2 und 2.4 wieso -1.)

So erhält der Treiber von zwei Seiten die Werte der Locators des interface Attribute an dem er „hängt“. Also die Position, Adresse, ID, ... mit der er bei seinem Vater bekannt ist. Die eine Seite von der ein Treiber die Locator Werte erhält ist das Array `cf_loc` der `cfdata` Datenstruktur. Diese Werte entsprechen denen in der Kernelkonfigurationsdatei angegebenen, werden also statisch zum Kompilationszeitpunkt vergeben. Die andere Seite ist die als `void *aux` and die `rfmatch` Funktion übergebene `attach_args` Datenstruktur. Sie wird vom Vater an das Kind weitergegeben, wenn der Kernel bootet. Sie enthält also die Locator Werte, die der tatsächlich vorhandenen Hardware entsprechen.

Mit der obigen `if` Abfrage prüft das Kind zwei Bedingungen ab: Zuerst ob als Locator ein Wildcard angegeben wurde. Falls ja „passt“ die Position, Adresse, ID, ... mit der es bei seinem Vater bekannt ist auf jeden Fall, es wird 1 zurückgegeben und der Kind Treiber „attached“. Wurde in der Kernelkonfigurationsdatei eine konkrete Position, Adresse, ID, ... für das Kind angegeben, muß dieses in seiner `match` Funktion überprüfen ob der Wert aus der Kernelkonfiguration mit dem aktuellen übereinstimmt. Wenn ja, isses gut und der Kind Treiber „attached“. Wenn nein passt die vorgefundenne Hardwarekonfiguration nicht zu der in der Kernelkonfigurationsdatei angegebenen und dann darf der Kind Treiber nicht

„attached“ werden. Also gibt die `match` Funktion 0 zurück. Notwendig ist diese Überprüfung um z.B. bestimmte Geräte auf bestimten Locators in der Kernelkonfiguration „festnageln“ zu können. Z.B. wie in:
`sd2 at scsibus1 target 6 lun 0`

Durch diesen Mechanismus ist es möglich sowohl Wildcards zu verwenden, als auch bestimmte Treiberinstanzen auf bestimmten Positionen „festzunageln“. Das ist ein weiterer Grund warum `config_found()` `config_search()` aufruft. `config_search()` iteriert über alle möglichen Kinder. Nur so ist es möglich das „richtige“ Kind zu finden, falls keine Wildcards verwendet wurden.

3.3.4 rf_attach()

```
void
rf_attach(struct device *parent, struct device *self, void *aux)
{
        struct rf_softc *rf_sc = (struct rf_softc *)self;
        struct rfc_attach_args *rfc_aa = (struct rfc_attach_args *)aux;
        struct rfc_softc *rfc_sc;
        struct disklabel *dl;

        rfc_sc = (struct rfc_softc *)rf_sc->sc_dev.dv_parent;
        rf_sc->sc_dnum = rfc_aa->dnum;
        rf_sc->sc_state = 0;
        rf_sc->sc_open = 0;
        rf_sc->sc_disk.dk_name = rf_sc->sc_dev.dv_xname;
        rf_sc->sc_disk.dk_driver = &rfdkdriver;
        disk_attach(&rf_sc->sc_disk);
        dl = rf_sc->sc_disk.dk_label;
```

Aus `autoconf(9)` Sicht nix besonderes, nur Initialisierung der `softc` Struktur und später benötigter Datenstrukturen. Die erste Zuweisung zeigt z.B. wie ein Kind-Gerät an die `softc` Struktur seines Vaters „herankommen“ kann. `disk_attach(9)` und die folgenden Zuweisungen initialisieren das Disklabel.

4 Der Treiberkern

In diesem Kapitel soll es um den eigentlichen Treiberkern gehen. Also was der Treiber an Funktionen und Datenstrukturen dem Rest des Kernels zur Verfügung stellen muß, damit Daten von und zur Hardware transferiert werden können und wie dies abläuft.

4.1 Datenstrukturen des Treibers

4.1.1 Datenstrukturen pro Controller

```
struct rfc_softc {
        struct device sc_dev; /* common device data */
        struct device *sc_childs[2]; /* child devices */
        struct evcnt sc_intr_count; /* Interrupt counter for statistics */
        struct buf *sc_curbuf; /* buf that is currently in work */
        bus_space_tag_t sc_iot; /* bus_space IO tag */
        bus_space_handle_t sc_ioh; /* bus_space IO handle */
        bus_dma_tag_t sc_dmat; /* bus_dma DMA tag */
        bus_dmamap_t sc_dmam; /* bus_dma DMA map */
        caddr_t sc_bufidx; /* current position in buffer data */
        int sc_curchild; /* child whose bufq is in work */
        int sc_bytesleft; /* bytes left to transfer */
        u_int8_t type; /* controller type, 1 or 2 */
};
```

sc_dev muß immer das erste Feld in einer `softc` Datenstuktur sein. Eine Vorgabe von `autoconf(9)`.

sc_childs enthält Zeiger auf die beiden möglichen Kindgeräte. (Jeder Controller kann max. zwei Laufwerke ansteuern.)

sc_intr_count Ereigniszähler für Interrupts zu statistischen Zwecken. Wird nicht unbedingt für die Funktion eines Treibers benötigt, sollte aber immer benutzt weren.

sc_iot, sc_ioh, sc_dmat, sc_dmah , , , sind die `bus_space(9)` und `bus_dma(9)` Tags und Handles mit denen der Treiber auf die Hardware zugreift.

sc_bufidx Ein Buffer kann größer als ein einzelner Sektor sein. Also muß sich der Treiber merken an welcher Stelle im Buffer er gerade arbeitet.

sc_curchild Enthält die Nummer des Kindgeräts, das gerade ein Kommando des Controllers abarbeitet.

sc_bytesleft Wieviele Bytes noch im Buffer übrig sind und noch von / zur Floppy zu transferieren sind.

type 1 oder 2 je nach dem ob es ein RX01 oder RX02 ist.

4.1.2 Datenstrukturen pro Laufwerk

```
#define RFS_DENS        0x0001    /* single or double density */
#define RFS_AD          0x0002    /* density auto detect */
#define RFS_NOTINIT     0x0000    /* not initialized */
#define RFS_PROBING     0x0010    /* density detect / verify started */
#define RFS_FBUF        0x0020    /* Fill Buffer */
#define RFS_EBUF        0x0030    /* Empty Buffer */
#define RFS_WSEC        0x0040    /* Write Sector */
#define RFS_RSEC        0x0050    /* Read Sector */
#define RFS_SMD         0x0060    /* Set Media Density */
#define RFS_RSTAT       0x0070    /* Read Status */
#define RFS_WDDS        0x0080    /* Write Deleted Data Sector */
#define RFS_REC         0x0090    /* Read Error Code */
#define RFS_IDLE        0x00a0    /* controller is idle */
#define RFS_CMDS        0x00f0    /* command mask */
#define RFS_OPEN_A      0x0100    /* partition a open */
#define RFS_OPEN_B      0x0200    /* partition b open */
#define RFS_OPEN_C      0x0400    /* partition c open */
#define RFS_OPEN_MASK   0x0f00    /* mask for open partitions */
#define RFS_OPEN_SHIFT  8         /* to shift 1 to get RFS_OPEN_A */
#define RFS_SETCMD(rf, state)   ((rf) = ((rf) & ~RFS_CMDS) | (state))


struct rf_softc {
        struct device sc_dev; /* common device data */
        struct disk sc_disk; /* common disk device data */
        struct bufq_state sc_bufq; /* queue of pending transfers */
        int sc_state; /* state of drive */
        u_int8_t sc_dnum; /* drive number, 0 or 1 */
};
```

sc_dev siehe oben.

sc_disk Jedes Gerät der Klasse `disk` muß diese Datenstruktur in seiner Datenstuktur haben. Der Kernel benötigt das zur Verwaltung des Plattenlaufwerke.

sc_bufq Die Bufferqueue des Laufwerks. Mehr dazu später.

sc_state Der Treiber merkt sich in dieser Variable in welchem Zustand das Laufwerk gerade ist um so die korrekte Abfolge nicht initialisiert, Initialisierung läuft, idle, Datentransfer, Sektor schreiben, idle, ... zu gewährleisten. Dem entsprechen die obigen symbolischen Definitionen.

sc_dnum Ist dies das erste oder zweite Laufwerk an diesem Controller?

4.2 Die notwendigen Funktionen

```
dev_type_open(rfopen);
dev_type_close(rfclose);
dev_type_read(rfread);
dev_type_write(rfwrite);
dev_type_ioctl(rfioctl);
dev_type_strategy(rfstrategy);
dev_type_dump(rfdump);
dev_type_size(rfsize);
```

Obige Macros sorgen nun dafür, dass der Kernel beim Linken nach folgenden Funktionen verlangt:

```
int rfopen(dev_t dev, int oflags, int devtype, struct proc *p);
int rfclose(dev_t dev, int fflag, int devtype, struct proc *p);
int rfread(dev_t dev, struct uio *uio, int ioflag);
int rfwrite(dev_t dev, struct uio *uio, int ioflag);
int rfioctl(dev_t dev, u_long cmd, caddr_t data, int fflag, struct proc *p);
void rfstrategy(struct buf *bp);
int rfdump(dev_t dev, daddr_t blkno, caddr_t va, size_t size);
int rfsize(dev_t dev);
```

Die `open, close, read, write, ioctl` Funktionen entsprechen direkt den aus dem User-Space bekannten Funktionen. Sprich ruft man z.B. ein `open(„/dev/rrf0c\, O_RDONLY, 0);` in einem Programm auf, führt das zum Aufruf von `rfopen()`. Die drei Funktionen `rfstrategy, rfdump, rfsize` sind für Kernelinterne Funktionen notwendig.

```
int rfc_sendcmd(struct rfc_softc *, int, int, int);
struct rf_softc* get_new_buf( struct rfc_softc *);
static void rfc_intr(void *);
```

Dies sind Hilfsfunktion des Treibers. Die erste, der Name legt es nahe, dient dazu ein Kommando an den Controller zu senden. Die letzte ist der Interrupthandler und bildet mit `rfstrategy()` den zentralen Dreh und Angelpunkt des Treibers. Die zweite ist eine Hilfsfunktion des Interrupthandlers.

```
int rfcprobedens(struct rfc_softc *, int);
```

Ist nur eine Hilfsfunktion zu Debuggingzwecken. Siehe auch die diesbezüglichen Kommentare in `rfc_attach`.

Ein kurzer Überblick über die prinzipiellen Abläufe innerhalb des Treibers, bevor wir uns in den kleinen, lästigen Details der einzelnen Funktionen verlieren: Um mit dem Gerät irgendetwas tun zu können, muß zuerst die `rfopen()` Funktion aufgerufen werden. Dies geschieht immer und ist essentiell für den Treiber, denn nur so kann er seinen internen Zustand initialisieren. Zu beachten ist dabei, dass die `open()` Funktion eines Treibers auch mehrfach hintereinander aufgerufen werden kann. Z.B. um verschiedene Partitionen auf einer Platte oder verschiedene Ports einer seriellen Multiportkarte zu öffnen (unterschiedliche Minordevicenumbers öffnen) oder um über eine Filehandle Daten zu transferrieren und über ein anderes Steueranweisungen via `ioctl` (gleiche Minordevicenumber öffnen). Entweder initiiert ein Prozess aus dem User-Space, der den Devicenode öffnet, oder aber auch der Kernel selbst, z.B. bei einem `mount(2)`, den Aufruf der `open()` Funktion.

Besonderheit dabei: Jedes mal wenn ein Benutzer-Prozess aus dem User-Space, der den Devicenode öffnet oder der Kernel selbst bei einem `mount(2)` oder im Zusammenhang mit `RAIDframe(9)` etc. wird die `open()` Funktion aufgerufen. Die `close()` Funktion wird aber erst aufgerufen, wenn der letzte Benutzer des Geräts ein `close()` ausführt. Beispiel: Drei Prozesse, A, B und C öffnen nacheinander in dieser Reihenfolge den selben Devicenode und halten ihn offen was zum dreimaligen Aufruf der `open()` Funktion führt. Dann schließt Prozess A den Devicenode - der Treiber kriegt nichts davon mit. Prozess C schließt den Devicenode - der Treiber erhält immer noch keine Nachricht. Erst wenn Prozess B den Devicenode schließt erfolgt der Aufruf der `close()` Funktion, da dieser Prozess als letzter den Devicenode offen hatte.

Datentransfers von und zur Hardware wickelt die `rfstrategy()` Funktion ab. Sie ist die „Anlaufstelle“ des Treibers für Datentransfers, die der Treiber dem Kernel bereitstellen muß. Bei jeden Aufruf erhält die `rfstrategy()` Funktion einen

Zeiger auf einen einzelnen *Buffer*. Diese Buffer sind die vom berühmt berüchtigten Buffercache verwendeten und beschreiben einen blockorientierten Datentransfer. Also welche Daten von wo aus dem RAM auf welche Position / Adresse auf welches Gerät geschrieben werden sollen, bzw. was von welchem Gerät an welche Stelle im RAM gelesen werden soll.

Die `strategy()` Funktion selbst führt typischerweise keine direkten I/O Operationen durch, sondern macht Plausibilitätsprüfungen und organisiert die I/O Operationen lediglich. Im wesentlichen bedeutet das, die Buffer in sogenannte Bufferqueues einzusortieren (daher auch *strategy*). Die Bufferqueues werden dann Buffer für Buffer von weiteren Routinen abgearbeitet. Diese Routinen, wieviele, was sie konkret tun und wie, sind sehr spezifisch für den einzelnen Treiber. Der Kernel macht dem Treiber hier keine Vorgaben.

Typischerweise gibt es mindestens eine Hilfsfunktion neben der `strategy()` Funktion, den Interrupthandler. I/O Geräte sind verglichen mit der CPU langsam. So kann man nicht einfach warten bis eine I/O Operation abgeschlossen ist. Vielmehr initiiert der Treiber eine Operation und der Kernel tut andere Dinge, z.B. einem Prozess Rechenzeit geben. Wenn die Hardware die I/O Operation beendet, erzeugt sie einen Interrupt. Die „normale“ Programmausführung wird also durch ein elektrisches Signal der Hardware unterbrochen und eine spezielle Routine des Kernels aufgerufen. Diese Routine stellt fest welches Stück Hardware den Interrupt ausgelöst hat und ruft den Interrupthandler des dafür zuständigen Treibers auf. Dieser Interrupthandler beendet dann die zuvor initiierte I/O Operation in dem Fehlercodes der Hardware (z.B. Lesefehler auf der Floppy) überprüft, Indizes in Buffern weitergerückt, der nächste Buffer aus der Queue genommen wird, etc.

4.2.1 `rfdump()`

... wird von Kernel benutzt um einen Cordedump auf die Platte / in den Swapspace zu kübeln, wenn er sich dazu gezwungen sieht, sich zu erbrechen. Ich halte es für unwahrscheinlich, dass eine Floppy von 0.5MB Kapazität dazu ausreicht. Also besteht diese Funktion aus einem schlichten `return( ENXIO);`. Siehe auch `errno(2)`.

4.2.2 `rfsize()`

(Swap-)Partitionsgröße in Einheiten von `DEV_BSIZE` zurückgeben?

4.2.3 `rfopen()`

```
rfopen(dev_t dev, int oflags, int devtype, struct proc *p)
```

```
{
        struct rf_softc *rf_sc;
        struct rfc_softc *rfc_sc;
        struct disklabel *dl;
        int unit;

        unit = DISKUNIT(dev);
        if (unit >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[unit]) == NULL) {
                return(ENXIO);
        }
```

Zuerst muß `rfopen()` mal feststellen ob das Gerät überhaupt vorhanden ist und falls ja besorgt es sich die `softc` Datenstruktur dieses Geräts. Der Weg von der Devicenumber zur `softc` Datenstruktur ist, wie man sehen kann, recht einfach - wenn man weiß wie. Das Makro `DISKUNIT()` ermittelt anhand der Minordevicenumber die wievielte Instanz des Gerätes mit der Devicenumber adressiert wurde. Siehe `sys/disklabel.h`. Dann kommt die `cfdriver` Datenstruktur ins Spiel. `config(8)` legt für jedes in der Kernelkonfigurationsdatei vorhandenne Gerät in `ioconf.[ch]` im Kernelkompilierverzeichnis eine Instanz dieser Datenstruktur an. Dabei gilt die Namenskonvention `<DEV>_cd`. Dabei steht `<DEV>` für den in der Kernelkonfigurationsdatei angegebenen Namen, in diesem Fall also `rf`. `cd` steht für *C*onfiguration*D*river. Die Typdefinition dieser Datenstruktur findet sich in `sys/device.h`. Das `cd_devs` Feld dieser Datenstruktur ist ein Array von Zeigern, von denen jeder auf die `softc` Datenstruktur einer Geräteinstanz zeigt. Dieses Array ist dynamisch, da ja auch zur Laufzeit des Kernels Geräte hinzugefügt und gelöscht werden können. Das `cd_ndevs` Feld enthält die Anzahl der Geräteinstanzen = Anzahl der Felder von `cd_devs`. Daneben gibt es noch `cd_name`, einen Zeiger auf einen String mit dem Gerätenamen und den Enum `cd_class`, was die Geräteklasse angibt.

```
        rfc_sc = (struct rfc_softc *)rf_sc->sc_dev.dv_parent;
        dl = rf_sc->sc_disk.dk_label;
        switch (DISKPART(dev)) {
                case 0:                 /* Part. a is single density. */
                        /* opening in single and double density is sensless */
                        if ((rf_sc->sc_state & RFS_OPEN_B) != 0 )
                                return(ENXIO);
                        rf_sc->sc_state &= ~RFS_DENS;
                        rf_sc->sc_state &= ~RFS_AD;
                        rf_sc->sc_state |= RFS_OPEN_A;
                break;
```

```
                case 1:                 /* Part. b is double density. */
                        /*
                         * Opening a singe density only drive in double
                         * density or simultaneous opening in single and
                         * double density is sensless.
                         */
                        if (rfc_sc->type == 1
                            || (rf_sc->sc_state & RFS_OPEN_A) != 0 )
                                return(ENXIO);
                        rf_sc->sc_state |= RFS_DENS;
                        rf_sc->sc_state &= ~RFS_AD;
                        rf_sc->sc_state |= RFS_OPEN_B;
                break;
                case 2:                 /* Part. c is auto density. */
                        rf_sc->sc_state |= RFS_AD;
                        rf_sc->sc_state |= RFS_OPEN_C;
                break;
                default:
                        return(ENXIO);
                break;
        }
```

Typischerweise kennzeichnet man den geöffneten Zustand in der `softc` Datenstruktur und verriegelt z.B. bei Wechselmedien den manuellen Medienauswurf. So kann man mehrere gleichzeitige Zugriffe auf ein Gerät unterbinden und festhalten, ob das Gerät in einem speziellen Modus geöffnet wurde. Der `rf(4)` Treiber macht davon gebrauch in dem er verschiedenen Minordevicenumbers d.h. Partitionen bei einem Disk-Treiber, einfache oder doppelte Schreibdichte zuordnet, bzw. die Schreibdichte automatisch anhand des Formats der eingelegten Diskette ermittelt.

```
        if ((rf_sc->sc_state & RFS_CMDS) == RFS_NOTINIT) {
                rfc_sc->sc_curchild = rf_sc->sc_dnum;
                /*
                 * Controller is idle and density is not detected.
                 * Start a density probe by issuing a read sector command
                 * and sleep until the density probe finished.
                 * Due to this it is impossible to open unformated media.
                 * As the RX02/02 is not able to format its own media,
                 * media must be purchased preformated. fsck DEC marketing!
                 */
```

```
                RFS_SETCMD(rf_sc->sc_state, RFS_PROBING);
                disk_busy(&rf_sc->sc_disk);
                if (rfc_sendcmd(rfc_sc, RX2CS_RSEC | RX2CS_IE
                    | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
                    | ((rf_sc->sc_state & RFS_DENS) == 0 ? 0 : RX2CS_DD),
                    1, 1) < 0) {
                        rf_sc->sc_state = 0;
                        return(ENXIO);
                }
                /* wait max. 2 sec for density probe to finish */
                if (tsleep(rf_sc, PRIBIO | PCATCH, "density probe", 2 * hz)
                    != 0 || (rf_sc->sc_state & RFS_CMDS) == RFS_NOTINIT) {
                        /* timeout elapsed and / or somthing went wrong */
                        rf_sc->sc_state = 0;
                        return(ENXIO);
                }
        }
```

Beim ersten Öffnen ist das Laufwerk nicht initialisiert, d.h. es muß ein Sektor von der Floppy gelesen werden um fest zu stellen ob das eingelegte Medium auch der gewünschten Schreibdichte entspricht, bzw. bei „auto density“ muß der Treiber die Schreibdichte zuerst mal herausfinden. Also geht der Treiber in den Zustand `RFS_PROBING` über und meldet die Floppy-Disk als „beschäftigt“[11]. Sodann erhält der Controller ein Kommando um einen Sektor zu lesen und es wird auf die (nicht) erfolgreiche Beendigung des Kommandos gewartet.

Problem dabei: Man kann nicht einfach eine Warteschleife (busy waiting) wie mit `DELAY()` in `rfc_match()` laufen lassen. Busy waiting an dieser Stelle bedeutet, dass die ganze Maschine komplett *steht*. Wir sind ja gerade im Kernel![12]

Die korrekte Lösung des Problems sind die Funktionen `tsleep(9)` und `wakeup(9)`. `rfopen()` wurde von irgend einem Prozess angestoßen und dieser Prozess muß warten bis die Operation beendet ist. Andere Prozesse können unterdessen unbehelligt weiter laufen. `tsleep(9)` markiert den Prozess, der die `rfopen()` Operation angestoßen hat, als „schlafend“ und weist den Scheduler an die anderen Prozesse an die CPU zu lassen bis die Schlafbedingung aufgehoben ist.

Diese Aufhebung erfolgt im Interrupthandler. Der Controller bekam ja ein Lesekommando mit Interrupt Enable (`RX2CS_IE`), d.h. der Controller erzeugt einen Interrupt sobald er mit dem Kommando fertig ist, was wiederum zum Aufruf des

[11] Die Benutzung von `disk_busy` ist keine zwingende Notwendigkeit gehört aber zum „guten Ton“ und dient der Erhebung von statistischen Daten, siehe `iostat(8)`.

[12] In `rfc_match()` ist das akzeptabel, da das ja nur in der Bootphase geschieht.

Interrupthandlers führt. Der Interrupthandler prüft das Ergebnis des Kommandos, verändert die Zustandsvariablen des Treibers entsprechend und teilt dem Scheduler durch `wakeup(9)` mit, dass die Operation beendet ist und der schlafende Prozess aufgeweckt werden kann. Sobald der Prozess das nächste mal Rechenzeit bekommt läuft er an der Stelle im Kernelmode in `rfopen()` weiter, an der `tsleep(9)` aufgerufen wurde. Somit ist an dieser Stelle zu prüfen ob der Interrupthandler den erfolgreichen Abschluß der Operation feststellen konnte. (Oder es einen Timeout gab...)

```
        /* disklabel. We use different fake geometries for SD and DD. */
        if ((rf_sc->sc_state & RFS_DENS) == 0) {
                dl->d_nsectors = 10;            /* sectors per track */
                dl->d_secpercyl = 10;           /* sectors per cylinder */
                dl->d_ncylinders = 50;          /* cylinders per unit */
                dl->d_secperunit = 501; /* sectors per unit */
                /* number of sectors in partition */
                dl->d_partitions[2].p_size = 500;
        } else {
                dl->d_nsectors = RX2_SECTORS / 2;  /* sectors per track */
                dl->d_secpercyl = RX2_SECTORS / 2; /* sectors per cylinder */
                dl->d_ncylinders = RX2_TRACKS;     /* cylinders per unit */
                /* sectors per unit */
                dl->d_secperunit = RX2_SECTORS * RX2_TRACKS / 2;
                /* number of sectors in partition */
                dl->d_partitions[2].p_size = RX2_SECTORS * RX2_TRACKS / 2;
        }
        return(0);
}
```

Treiber für partitionierbare Medien sollten sich an dieser Stelle auch um das Auslesen des `disklabel(5,9)`s kümmern. Da wir kein `disklabel(5,9)` auf RX01/02 Floppies unterstützen erzeugen wir ein pseudo `disklabel(5,9)`.

4.2.4 `rfclose()`

```
int
rfclose(dev_t dev, int fflag, int devtype, struct proc *p)
{
        struct rf_softc *rf_sc;
        int unit;
```

```
        unit = DISKUNIT(dev);
        if (unit >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[unit]) == NULL) {
                return(ENXIO);
        }
        if ((rf_sc->sc_state & 1 << (DISKPART(dev) + RFS_OPEN_SHIFT)) == 0)
                panic("rfclose: can not close on non-open drive %s "
                    "partition %d", rf_sc->sc_dev.dv_xname, DISKPART(dev));
        else
                rf_sc->sc_state &= ~(1 << (DISKPART(dev) + RFS_OPEN_SHIFT));
        if ((rf_sc->sc_state & RFS_OPEN_MASK) == 0)
                rf_sc->sc_state = 0;
        return(0);
}
```

Falls ein nicht zuvor geöffnetes Gerät geschlossen werden soll ist dies ein erheblicher Fehler, daher gibt es in dem Fall einen Kernelpanic. Im anderen Fall wird nur das Bit zurückgesetzt, das den geöffneten Zustand der Partition markiert. Die letzte if-Abfrage prüft, ob alle Partitionen geschlossen sind und wenn ja sind die von der `rfopen` vorgenommenen Verriegelungen aufzuheben und der Zustand des Geräts in der `softc` Datenstruktur ist zurückzusetzen.

4.2.5 `rfread()` und `rfwrite()`

Bestehen aus:

```
return( physio( rfstrategy, NULL, dev, B_READ, minphys, uio));
bzw.
return( physio( rfstrategy, NULL, dev, B_WRITE, minphys, uio));
```

Wenn ein Prozess via `read(2)/write(2)` Daten auf den Characterdevicenode schreiben oder davon lesen will, nimt die Mimik hinter `read(2)/write(2)` diesen Wunsch entgegen und dröselt die Zugehörigkeit des Dateideskriptors zu dem Devicenode auf. Der bei `read(2)/write(2)` angegebene Buffer wird von der `read(2)/write(2)` Mimik zerlegt bis ein Satz an Zeigern auf Pages im pysikalischen RAM übrig bleibt (plus Offset und Länge innerhalb der Page). Diese Zeiger landen dann in der `uio` Struktur. Die Funktion `physio()` baut diese Zeiger auf Pages in der `uio` Struktur in Filesystembuffer um, wie sie vom Buffercache verwendet werden (`struct buf`) und ruft die angegebene `strategy()` Funktion auf um die Filesystembuffer einen nach dem anderen an den Treiber zu übergeben. Siehe `physio(9)` und `sys/kern/kern_physio.c`. Zugriffe über das Characterdevice gehen `nicht` über den Buffercache, sondern `physio(9)` transferiert die Daten unmittelbar zwischen dem Speicher des Prozesses und dem Treiber.

4.2.6 `rfstrategy()`

Wie oben schon kurz angesprochen ist diese Funktion die zentrale Anlaufstelle des Treibers für Datentransfers. Sie nimmt die Buffer entgegen und füllt oder leert sie. `rfstrategy()` verursacht selbst keine IO Operationen. Viel mehr schreibt `rfstrategy()` diese IO-Requests in eine (oder mehrere) Queues. Wobei es die Kunst des Treibers ist, diese Queues umzusortieren, um z.B. Bewegungen des Schreib-/Lesekopfes zu minimieren. Geleert werden diese Queues von dem Interrupthandler. Jedes mal, wenn die Hardware eine IO Operation abgeschlossen hat, generiert sie einen Interrupt. Der Interrupthandler registriert diese Operation als abgeschlossen und entfernt sie aus der Queue. Sodann initiiert der Interrupthandler die nächste IO Operation, die in der Queue ansteht, ...

```
rfstrategy(struct buf *buf)
{
        struct rf_softc *rf_sc;
        struct rfc_softc *rfc_sc;
        int i;

        i = DISKUNIT(buf->b_dev);
        if (i >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[i]) == NULL) {
                buf->b_flags |= B_ERROR;
                buf->b_error = ENXIO;
                biodone(buf);
                return;
        }
```

Das aus `rfopen()` bekannte Spiel um an die `rf_sc softc` Datenstruktur zu gelangen. Die Fehlerbehandlung ist allerdings anders. Ein Buffer kann auch als Arbeitsauftrag verstanden werden. Verschiedene Buffer sind unabhängig von einander. Man bedenke diese einzige `rfstrategy()` Funktion ist die Anlaufstelle für alle `rf(4)` Geräteinstanzen. Ein Buffer kann einen Arbeitsauftrag für das erste Laufwerk enthalten, der erfolgreich abgeschlossen werden kann. Ein anderer Buffer bezieht sich auf ein Laufwerk, das physikalisch nicht vorhanden ist und daher fehlschlagen muß. Oder ein Buffer wird später über einen defekten Sektor stolpern und andere Buffer sind davon nicht betroffen.

Auch der Zeitpunkt an dem ein Buffer „fertig“ ist, ist unbekannt. Ist das Laufwerk physikalisch nicht vorhanden ist das schon beim Aufruf von `rfstrategy()` feststellbar (genau das prüft obige if-Abfrage). Womöglich wird der Buffer aber zuerst weit hinten in der Bufferqueue einsortiert. In diesem Fall ist erst sehr viel später, nach dem `rfstrategy()` den Buffer längst „gefressen“ und mit `return` zurückgesprungen ist, bekannt das der Buffer abgearbeitet ist. Oder anders ausgedrückt: Die Beendigung eines Buffers ist asynchron zur `rfstrategy()` Funktion. Daher muß dem Kernel mit der Funktion `biodone(9)` signalisiert werden wann der Buffer fertig ist. Trat dabei ein Fehler auf setzt man wie oben zu sehen ein Fehlerflag und gibt einen entsprechenden Fehlercode weiter.

```
rfc_sc = (struct rfc_softc *)rf_sc->sc_dev.dv_parent;
/* We are going to operate on a non open dev? PANIC! */
if ((rf_sc->sc_state & 1 << (DISKPART(buf->b_dev) + RFS_OPEN_SHIFT))
    == 0)
        panic("rfstrategy: can not operate on non-open drive %s "
            "partition %d", rf_sc->sc_dev.dv_xname,
            DISKPART(buf->b_dev));
```

Der Kommentar sagt es... ;-)

```
if (buf->b_bcount == 0) {
        biodone(buf);
        return;
}
```

Eine kleine Optimierung. Wenn `buf->b_bcount == 0` ist, ist auch nix zu tun, also sind wir schnell fertig mit dem Buffer.

```
/*
 * BUFQ_PUT() operates on b_rawblkno. rfstrategy() gets
 * only b_blkno that is partition relative. As a floppy does not
 * have partitions b_rawblkno == b_blkno.
 */
buf->b_rawblkno = buf->b_blkno;
/*
 * from sys/kern/subr_disk.c:
 * Seek sort for disks.  We depend on the driver which calls us using
 * b_resid as the current cylinder number.
 */
i = splbio();
if (rfc_sc->sc_curbuf == NULL) {
        rfc_sc->sc_curchild = rf_sc->sc_dnum;
        rfc_sc->sc_curbuf = buf;
        rfc_sc->sc_bufidx = buf->b_un.b_addr;
        rfc_sc->sc_bytesleft = buf->b_bcount;
        rfc_intr(rfc_sc);
```

```
        } else {
                buf->b_resid = buf->b_blkno / RX2_SECTORS;
                BUFQ_PUT(&rf_sc->sc_bufq, buf);
                buf->b_resid = 0;
        }
        splx(i);
        return;
}
```

Nehmen wir einmal an, die `rfstrategy()` Funktion erhält kurz hinterander drei Buffer. Der erste liest einen Sektor von Spur 1, der zweite liest einen Sektor von der letzten Spur und der dritte liest einen Sektor irgendwo aus der Mitte. Würde die `rfstrategy()` Funktion diese Buffer dumm-stur in dieser Reihenfolge in die Bufferqueue stellen, müßte der Controller den Schreib-/Lesekopf zuerst auf Spur 1, dann quer über die ganze Plattenoberfläche auf die letzte Spur und zuletzt zurück in die Mitte bewegen. Bewegungen des Schreib-/Lesekopfes sind *langsam*, also „teuer“. Man muß eine Minimierung dieser Bewegungen anstreben. Daher bietet es sich an die Bufferqueue so zu sortieren, dass zuerst der erste Buffer, dann der dritte und zuletzt der zweite abgearbeitet wird. Das Lesen des Sektors in der Mitte erfolgt sozusagen „nebenbei“ als Zwischenstopp auf dem Weg zur letzten Spur. Glücklicherweise muß man sich in einem Treiber keine Gedanken über diese Sortiererei machen sondern überläßt sie der Funktion `BUFQ_PUT()`. `BUFQ_PUT()` erwartet in dem Feld `b_resid` die Spur- bzw. Zylindernummer (die ja abhängig von der Plattengeometrie ist) um diese Optimierung nach Zylindernummern durchführen zu können. `BUFQ_PUT()` fügt den Buffer in die Bufferqueue ein und sortiert sie dabei entsprechen neu.

Diese ganze Sortiererei kann man sich aber sparen, wenn der Controller gerade idle ist. Dann ist `rfc_sc->sc_curbuf == NULL`. `rfc_sc->sc_curbuf` zeigt, wie der Name nahe legt, auf den Buffer, der gerade in Arbeit ist. Ist der Controller idle, so wird der Buffer als der aktuelle gesetzt, und die anderen damit zusammenhängenden Variablen der `softc` Datenstruktur initialisiert. Kommen noch weitere Buffer wärend der erste in `rfc_sc->sc_curbuf` in Arbeit ist, landen diese in der Bufferqueue. Der Aufruf von `rfc_intr()` stößt den eigentlichen Datentransfer an. Alles weitere erledigt die `rfc_intr()` Funktion im Interruptkontext.

Interruptkontext, ein gutes Stichwort. Was ist das eigentlich und gibt es noch andere Kontexte? Vereinfacht gibt es drei Kontexte in Unix:

Userkontext: Ein normaler Userprozess hat die CPU. Diese Zeit entspricht der „xx% user“ Angabe in `top(1)` oder `time(1)`.

Kernelkontext: Ein Userprozess hat einen Systemaufruf getätigt. Es wird Kernelcode im privilegierten Modus abgearbeitet, aber immer noch auf „Rechnung“ des Prozesses. Diese Zeit entsprcht der „xx% system“ Angabe in `top(1)` oder `time(1)`.

Interruptkontext: Wärend sich die CPU im User- oder Kernelkontext befindet, hat die Hardware mit einem elektrischen Signal die Unterbrechung der aktuellen Codeausführung angefordert. Diese Anforderung hat Priorität über den User- oder Kernelkontext. Der momentane Zustand der CPU wird gesichert und der Interrupthandler aufgerufen. Dieser bedient die Hardware und bei seinem Ende wird der CPU-Zustand wiederhergestellt und die Codeausführung da fortgesetzt wo sie unterbrochen wurde. Code des User- oder Kernelkontextes merkt davon erstmal nichts.

Findige Leser erkennen hier schon das Problem: Ein Interrupt ist ein asynchrones Ereignes, das jederzeit auftreten kann. Also z.B. auch gerade in dem Augenblick in dem die `rfstrategy()` Funktion die Bufferqueue manipuliert (neue Buffer hinzufügt). Der Interrupthandler des `rf(4)` Treibers manipuliert aber ebenfalls die Bufferqueue (entfernt abgearbeitete Buffer). Sollte dies geschehen ist die Bufferqueue in einem inkonsistenten Zustand. Es knallt, Kernelpanic, Absturz, Ende des Spiels.

Daher muß die `rfstrategy()` Funktion mit `splbio(9)` sicherstellen, dass ihr kein Interrupt „dazwischefunken“ kann. Diese Funktion blockiert alle Interrupts bis sie wieder mit `splx(9)` frei gegeben werden. Die Zeit in der Interrupts gesperrt sind sollte so gering wie möglich sein, denn es sind dann nicht nur Interrupts für dieses Gerät gesperrt, sondern *alle*. Bei großzügigem Interruptsperren könnte ein Interrupt für ein anderes Gerät unnötig verzögert werden, was den I/O Durchsatz unnötig verringert.

Um noch etwas genauer zu werden: Es gibt verschiedene Interrupt Prioritäten. `IPL_BIO` z.B. ist relativ niedrig und wird für Block I/O orientierte Geräte wie Platten und Floppies benutzt. Diese Geräte sind ohnehin langsam und haben i.d.R. größere Buffer in Hardware auf dem Controller. Es is also nicht so schlimm, wenn solch ein Gerät etwas länger auf die Bedienung eines Interrupts warten muß. Netzwerkkarten benutzen `IPL_NET` was eine Priorität über `IPL_BIO` ist. Netzwerkkarten sind „schneller“ als Platten (keine Mechanik) und Bufferoverruns bei Netzwerkkarten sind „schmerzhaft“, sie können z.B. TCP retries == erhöhte Netzlast bei verringertem Durchsatz auslösen. So ist es mit dieser Priorisierung möglich, dass ein Interrupt von einer Netzwerkkarte den Interrupthandler eines Plattentreibers unterbricht, der Interrupthandler des Netzwerkkartentreibers wird abgearbeitet und dann geht es weiter mit dem Interrupthandler des Plattentreibers.

Es ist also wichtig Interrupts nicht nur kurz zu sperren, sondern auch mit der richtigen Priorität. Ist die Priorität zu gering kann der eigene Interrupthandler „dazwischefunken“, ist die Priorität zu hoch kann der Durchsatz anderer Geräte leiden. Siehe auch `spl(9)` zum Zusammenspiel der verschiedenen Prioritäten.

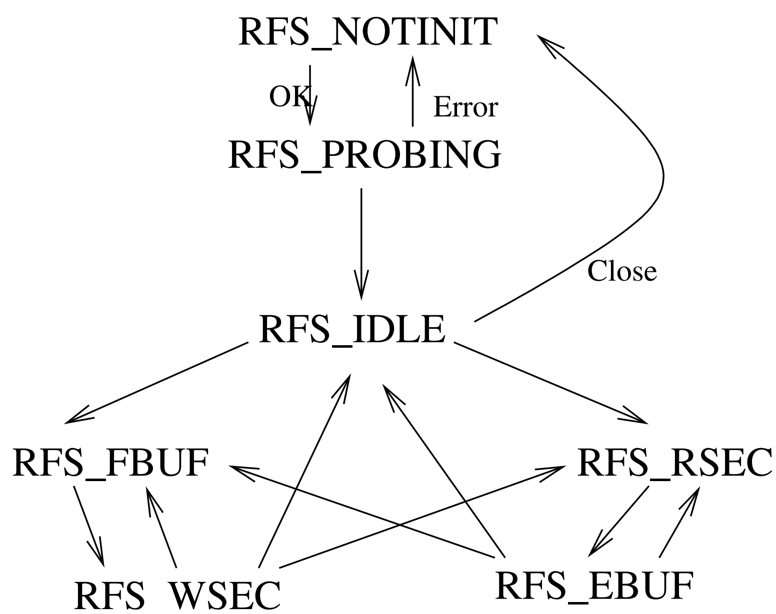

Abbildung 3: Diagramm der `rf(4)` internen Zustände und Übergänge.

4.2.7 `rfc_intr()`

`rfc_intr()` ist das eigentliche Arbeitspferd des Treibers, das die Buffer füllt und leert und so die Bufferqueues abarbeitet. Um eine Operation durchzuführen muß der Treiber / das Laufwerk bestimmte Zustände in einer bestimmten Reihenfolge durchlaufen. Beim ersten `open()` ist der Zustand `RFS_NOTINIT` und wird sodann auf `RFS_PROBING` gesetzt bis die Schreibdichte des eingelegten Mediums festgestellt ist. Dann geht der Treiber in den Zustand `RFS_IDLE` über bis ein Buffer mit einem Lese- oder Schreibkommando kommt.

Zum Lesen oder Schreiben eines Sektors müssen zwei Kommandos an den Controller gesendet werden. Zuerst einen Lesekommando `RX2CS_RSEC` (ReadSECtor) um den Sektor in den controllerinternen Buffer zu lesen und dann ein DMA-Kommando `RX2CS_EBUF` (EmptyBUFfer) um die Daten vom Controller in das RAM des Rechners zu transferieren. Sind noch nicht alle Daten des Buffers aus der Bufferqueue übertragen, sendet der Treiber das nächste `RX2CS_RSEC` Kommando, dann `RX2CS_EBUF` usw. bis der Buffer aus der Bufferqueue abgearbeitet ist. Analog beim Schreiben: Zuerst die Daten vom RAM des Rechners per DMA in den controllerinternen Buffer schaufeln `RX2CS_FBUF` (FillBUFfer) und mit einem zweiten Kommando `RX2CS_WSEC` (WriteSECtor) den Inhalt des controllerinternen Buffers auf das Medium schreiben und dann das nächste `RX2CS_FBUF` Kommando ... bis der nächste Buffer aus der Bufferqueue dran ist. Diesen Kommandos entsprechen die Zustände RFS_RSEC, RFS_EBUF, RFS_FBUF, RFS_WSEC. Von den vier weiteren Kommandos RX2CS_SMD, RX2CS_RSTAT, RX2CS_WDDS, RX2CS_REC und den assoziierten Zuständen macht der Treiber derzeit keinen Gebrauch. Nach dem Zustand RFS_EBUF muß nicht immer RFS_RSEC folgen. Ist der Buffer abgearbeitet und die Bufferqueue leer geht der Treiber in den Zustand RFS_IDLE. Ist die Bufferqueue nicht leer, sondern es folgt ein neuer Buffer, dessen Daten geschrieben werden müssen erfolgt ein Übergang von RFS_EBUF zu RFS_FBUF. Analog ergeben sich Zustandsübergänge von RFS_WSEC zu RFS_IDLE oder RFS_RSEC. Die Funktion rfc_sendcmd() vereinfacht das Senden der Kommandos ein wenig.

Auf Grund der Länge dieser Funktion hier nur die Erläuterung der grundsätzlichen Funktion und einige Ausschnitte von besonderer Bedeutung: Die Funktion besteht aus zwei switch Anweisungen. Die erste kümmert sich darum nach dem letzten Kommando / Zustand aufzuräumen und die letzte Operation abzuschließen. Die zweite leitet das nächste Kommando ein. Beide switch Anweisungen sind in eine Schleife eingebettet. Diese wird abgebrochen sobald entweder ein neues Kommando erfolgreich an den Controller gesendet wurde oder die Bufferqueue leer ist. Trat ein Fehler auf springt der Interrupthandler mittels continue an den Anfang der Schleife und versucht den nächsten Buffer zu bearbeiten (in anderen Treibern findet man in solchen Fällen gelegentlich ein goto. Da ich zwar Spaghetti, aber keinen Spaghetticode, mag ist die Lösung mit der Schleife vorzuziehen).

Folgend die Programmausschnitte die bei einem Lesevorgang ausgehend vom Zustand RFS_IDLE durchlaufen werden. Beginnend mit der get_new_buf() Hilfsfunktion.

```
struct rf_softc*
get_new_buf( struct rfc_softc *rfc_sc)
{
    struct rf_softc *rf_sc;
    struct rf_softc *other_drive;

    rf_sc = (struct rf_softc *)rfc_sc->sc_childs[rfc_sc->sc_curchild];
    rfc_sc->sc_curbuf = BUFQ_GET(&rf_sc->sc_bufq);
    if (rfc_sc->sc_curbuf != NULL) {
        rfc_sc->sc_bufidx = rfc_sc->sc_curbuf->b_un.b_addr;
        rfc_sc->sc_bytesleft = rfc_sc->sc_curbuf->b_bcount;
    } else {
        RFS_SETCMD(rf_sc->sc_state, RFS_IDLE);
        other_drive = (struct rf_softc *)
            rfc_sc->sc_childs[ rfc_sc->sc_curchild == 0 ? 1 : 0];
```

```
        if (other_drive != NULL
            && BUFQ_PEEK(&other_drive->sc_bufq) != NULL) {
            rfc_sc->sc_curchild = rfc_sc->sc_curchild == 0 ? 1 : 0;
            rf_sc = other_drive;
            rfc_sc->sc_curbuf = BUFQ_GET(&rf_sc->sc_bufq);
            rfc_sc->sc_bufidx = rfc_sc->sc_curbuf->b_un.b_addr;
            rfc_sc->sc_bytesleft = rfc_sc->sc_curbuf->b_bcount;
        } else
            return(NULL);
    }
    return(rf_sc);
}
```

Ist ein Buffer abgearbeitet ruft `rfc_intr` diese Funktion auf um den nächsten zu bearbeitenden Buffer zu erhalten. Der Controller hat zwei Laufwerke. Zuerst prüft diese Funktion ob noch Buffer in der Bufferqueue des aktuellen Laufwerks (`rfc_sc->sc_curchild`) vorhanden sind. Wenn ja kommt der nächste Buffer aus dieser Queue dran. Ist die Bufferqueue leer wird das Laufwerk als idle markiert und es wird geprüft ob in der Bufferqueue des anderen Laufwerks (`other_drive`) Buffer bereit liegen. Falls nein erfolgen keine weiteren Aktionen, es ist ja auch nix zu tun. Falls Buffer vorhanden sind schaltet die Funktion das aktuelle Laufwerk (`rfc_sc->sc_curchild`) um und leitet die Bearbeitung des nächsten Buffers ein. Der Rückgabewert der Funktion ist `NULL` falls beide Bufferqueues leer sind oder ein Zeiger auf die `softc` Struktur des Laufwerks, dessen Bufferqueue jetzt bearbeitet wird.

```
        /* erste switch Anweisung */
        case RFS_IDLE:  /* controller is idle */
            if (rfc_sc->sc_curbuf->b_bcount
                % ((rf_sc->sc_state & RFS_DENS) == 0
                ? RX2_BYTE_SD : RX2_BYTE_DD) != 0) {
                /*
                 * can only handle blocks that are a multiple
                 * of the physical block size
                 */
                rfc_sc->sc_curbuf->b_flags |= B_ERROR;
            }
            RFS_SETCMD(rf_sc->sc_state, (rfc_sc->sc_curbuf->b_flags
                & B_READ) != 0 ? RFS_RSEC : RFS_FBUF);
            break;
```

Die if-Abfrage macht eine Plausibilitätsprüfung, siehe Kommentar. Das Makro `RFS_SETCMD` vereinfacht das Setzen des Zustands der `rf_sc->sc_state` Variablen. Je nachdem ob der aktuelle Buffer eine Lese- oder Schreibanforderung enthält entsprechend `RFS_RSEC` oder `RFS_FBUF`.

```
/* zweite switch Anweisung */
case RFS_RSEC:  /* Read Sector */
    i = (rfc_sc->sc_curbuf->b_bcount - rfc_sc->sc_bytesleft
        + rfc_sc->sc_curbuf->b_blkno * DEV_BSIZE) /
        ((rf_sc->sc_state & RFS_DENS) == 0
        ? RX2_BYTE_SD : RX2_BYTE_DD);
    if (i > RX2_TRACKS * RX2_SECTORS) {
        rfc_sc->sc_curbuf->b_flags |= B_ERROR;
        break;
    }
    disk_busy(&rf_sc->sc_disk);
    if (rfc_sendcmd(rfc_sc, RX2CS_RSEC | RX2CS_IE
        | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
        | ((rf_sc->sc_state& RFS_DENS) == 0 ? 0 : RX2CS_DD),
        i % RX2_SECTORS + 1, i / RX2_SECTORS) < 0) {
        disk_unbusy(&rf_sc->sc_disk, 0, 1);
        rfc_sc->sc_curbuf->b_flags |= B_ERROR;
    }
    break;
```

Die erste Anweisung berechnet in der Variablen `i` die logische Blocknummer die nun von der Floppy gelesen werden soll. Dabei ist zu beachten, dass das RX02 Laufwerk nicht wie sonst üblich 512 Bytes pro Sektor (`DEV_BSIZE`, die Einheit von `buf->b_blkno`) sondern 128 Bytes pro Sektor (`RX2_BYTE_SD`) in einfacher und 256 Byte (`RX2_BYTE_DD`) in doppelter Schreibdichte hat. Die if-Anweisung prüft ob der vom Buffer angeforderte Sektor die Kapazität der Floppy übersteigt und wenn ja wird das Fehlerflag im Buffer gesetzt und die Operation abgebrochen. Zu `disk_busy()` siehe die Erläuterungen in Abschnitt 4.2.3.

Ja, und dann bekommt der Controller endlich via `rfc_sendcmd()` das Kommando zum Lesen eines Sektors (`RX2CS_RSEC`) mit Interruptenablebit `RX2CS_IE` natürlich. Falls das daneben geht setzt am Ende der Schleife um die beiden `switch` Anweisungen die Fehlerbehandlung ein:

Die `rfc_intr()` Funktion prüft nach jeder der beiden `switch` Anweisungen ob das Fehlerflag gesetzt ist und bringt den Treiber im Fehlerfall in einen definierten Zustand:

```
if ((rfc_sc->sc_curbuf->b_flags & B_ERROR) != 0) {
```

```
            /*
             * An error occured while processing this buffer.
             * Finish it and try to get a new buffer to process.
             * Return if there are no buffers in the queues.
             * This loops until the queues are empty or a new
             * action was successfully scheduled.
             */
            rfc_sc->sc_curbuf->b_resid = rfc_sc->sc_bytesleft;
            rfc_sc->sc_curbuf->b_error = EIO;
            biodone(rfc_sc->sc_curbuf);
            rf_sc = get_new_buf( rfc_sc);
            if (rf_sc == NULL)
                return;
            continue;
        }
```

So. Jetzt ist der Interrupthandler am Ende und es geht im alten Kontext weiter... Bis irgendwann der Controller mit dem Kommando fertig ist und einen Interrupt auslöst. Dann gehts weiter im Interrupthandler des `rf(4)` Treibers:

```
        /* erste switch Anweiung */
        case RFS_RSEC:  /* Read Sector */
            disk_unbusy(&rf_sc->sc_disk, 0, 1);
            /* check for errors */
            if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh,
                RX2CS) & RX2CS_ERR) != 0) {
                /* should do more verbose error reporting */
                printf("rfc_intr: Error reading sector: %x\n",
                    bus_space_read_2(rfc_sc->sc_iot,
                    rfc_sc->sc_ioh, RX2ES) );
                rfc_sc->sc_curbuf->b_flags |= B_ERROR;
            }
            RFS_SETCMD(rf_sc->sc_state, RFS_EBUF);
            break;
```

Zuerst mal dem Kernel mitteilen, dass das Laufwerk nicht länger beschäftigt ist, das bisher 0 Bytes übertragen wurden und es ein Lesekommando war. Die if-Abfrage prüft das Fehlerflag (`RX2CS_ERR`) im CSR des Controllers und bricht den Vorgang gegebenenfalls in bekannter Manier ab. An dieser Stelle könnte mit den Kommandos `RX2CS_RSTAT` und `RX2CS_REC` eine detailiertere Fehlerdiagnose einsetzen, aber das sparen wir uns aus Komplexitätsgründen. ;-)

```
/* zweite switch Anweiung */
case RFS_EBUF:  /* Empty Buffer */
    i = bus_dmamap_load(rfc_sc->sc_dmat, rfc_sc->sc_dmam,
        rfc_sc->sc_bufidx, (rf_sc->sc_state & RFS_DENS) == 0
        ? RX2_BYTE_SD : RX2_BYTE_DD,
        rfc_sc->sc_curbuf->b_proc, BUS_DMA_NOWAIT);
    if (i != 0) {
        printf("rfc_intr: Error loading dmamap: %d\n",
        i);
        rfc_sc->sc_curbuf->b_flags |= B_ERROR;
        break;
    }
    disk_busy(&rf_sc->sc_disk);
    if (rfc_sendcmd(rfc_sc, RX2CS_EBUF | RX2CS_IE
        | ((rf_sc->sc_state & RFS_DENS) == 0 ? 0 : RX2CS_DD)
        | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
        | ((rfc_sc->sc_dmam->dm_segs[0].ds_addr
        & 0x30000) >>4), ((rf_sc->sc_state & RFS_DENS) == 0
        ? RX2_BYTE_SD : RX2_BYTE_DD) / 2,
        rfc_sc->sc_dmam->dm_segs[0].ds_addr & 0xffff) < 0) {
        disk_unbusy(&rf_sc->sc_disk, 0, 1);
        rfc_sc->sc_curbuf->b_flags |= B_ERROR;
        bus_dmamap_unload(rfc_sc->sc_dmat,
        rfc_sc->sc_dmam);
    }
    break;
```

Ein anstehender DMA-Transfer ist bei dem `bus_dma(9)` System anzumelden. Der UniBus hat einen 18 Bit Adressraum und der QBus (zumindest bei VAXen) hat 22 Bit Adressraum. D.h. ein UniBus / QBus Gerät kann nicht den gesamten Adressraum der CPU erreichen sondern nur einen kleinen Ausschnitt. Ähnlich wie der ISA Bus mit seinen 24 Bit Adressraum. Da die Leute bei DEC allerdings nachgedacht haben, als sie die VAX entwarfen, schufen sie eine Abhilfe für dieses Problem (im Gegensatz zu den IBM Technikern die den PeeCee/AT verbrochen haben). Beim UniBus / QBus sitzt im Bus Adapter, also zwischen UniBus / QBus Adressraum und dem CPU Adressraum eine MMU. Diese MMU kann so programmiert werden, dass beliebige UniBus / QBus Adressen auf beliebige CPU Adressen umgesetzt werden können. So kann ein UniBus / QBus Gerät den gesammten CPU Adressraum per Busmaster-DMA erreichen, vorausgesetzt die Busadapter MMU ist richtig programmiert. Gibt es eine solche MMU im Busadapter nicht bleibt nur der aufwändige Weg über „Bounce Buffer“ wie bei

ISA, siehe [Tho]. Doch um all das müssen wir uns als Treiberprogrammierer nicht kümmern. `bus_dmamap_load(9)` erledigt das.

Sofern das `bus_dma(9)` System dem Treiber eine DMA-Map gegeben hat, kann der Treiber die Floppy als beschäftigt melden und die eigentliche DMA Operation mit dem `RX2CS_EBUF` Kommando initiieren. Und erneut warten wir auf den nächsten Interrupt...

```
/* erste switch Anweiung */
case RFS_EBUF:  /* Empty Buffer */
    i = (rf_sc->sc_state & RFS_DENS) == 0
        ? RX2_BYTE_SD : RX2_BYTE_DD;
    disk_unbusy(&rf_sc->sc_disk, i, 1);
    bus_dmamap_unload(rfc_sc->sc_dmat, rfc_sc->sc_dmam);
    /* check for errors */
    if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh,
        RX2CS) & RX2CS_ERR) != 0) {
        /* should do more verbose error reporting */
        printf("rfc_intr: Error while DMA: %x\n",
            bus_space_read_2(rfc_sc->sc_iot,
            rfc_sc->sc_ioh, RX2ES));
        rfc_sc->sc_curbuf->b_flags |= B_ERROR;
        break;
    }
```

Dieser Aufruf von `rfc_intr()` beendet den Transfer. `disk_unbusy(9)` gibt der Statistikabteilung bekannt wieviele Bytes gelesen wurden, die DMA-Map wird frei gegeben und die übliche Fehlerüberprüfung.

```
    if (rfc_sc->sc_bytesleft > i) {
        rfc_sc->sc_bytesleft -= i;
        rfc_sc->sc_bufidx += i;
```

Der Buffer ist noch nicht leer, also nur die Zeiger weiterrücken und und das nächste `RFS_RSEC` Kommando vorbereiten...

```
    } else {
        biodone(rfc_sc->sc_curbuf);
        rf_sc = get_new_buf( rfc_sc);
        if (rf_sc == NULL)
            return;
    }
    RFS_SETCMD(rf_sc->sc_state,
```

```
                (rfc_sc->sc_curbuf->b_flags & B_READ) != 0
                ? RFS_RSEC : RFS_FBUF);
            break;
```

Aha. Der Buffer ist komplett und erfolgreich abgearbeitet, also dem Rest des Kernels via `biodone(9)` die freudige Nachrich verkünden. Sodann ist zu prüfen ob noch weitere Buffer in der Bufferqueue stehen und wenn ja wird der nächste abgearbeitet. Ist kein Buffer mehr in der Bufferqueue dieses Laufwerks, geht das Laufwerk in den Idle Zustand und der Treiber schaltet auf das andere Laufwerk um, falls in dessen Bufferqueue Buffer eingetragen wurden wärend sich die Bufferqueue des ersten Laufwerks in Arbeit befand.

4.2.8 **`rfioctl()`**

Diese Funktion ist die Anlaufstelle für `ioctl(2)` Aufrufe auf das Gerät. In diesem Fall sind nur die IOCTLs zum Lesen des `disklabel(5,9)` unbedingt notwendig, andere IOCTLs sind in diesem Treiber nicht notwendig bzw. sinnvoll. Das RX02 Laufwerk kann nicht von der Software verriegelt oder das Medium ausgeworfen werden, die Hardware ist nicht in der Lage ein low-level Format zu machen...

```
int
rfioctl(dev_t dev, u_long cmd, caddr_t data, int fflag, struct proc *p)
{
        struct rf_softc *rf_sc;
        int unit;

        unit = DISKUNIT(dev);
        if (unit >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[unit]) == NULL) {
                return(ENXIO);
        }
        /* We are going to operate on a non open dev? PANIC! */
        if (rf_sc->sc_open == 0) {
                panic("rfstrategy: can not operate on non-open drive %s (2)",
                    rf_sc->sc_dev.dv_xname);
        }
        switch (cmd) {
        /* get and set disklabel; DIOCGPART used internally */
        case DIOCGDINFO: /* get */
                memcpy(data, rf_sc->sc_disk.dk_label,
                    sizeof(struct disklabel));
                return(0);
```

```
        case DIOCSDINFO: /* set */
                return(0);
        case DIOCWDINFO: /* set, update disk */
                return(0);
        case DIOCGPART:  /* get partition */
                ((struct partinfo *)data)->disklab = rf_sc->sc_disk.dk_label;
                ((struct partinfo *)data)->part =
                    &rf_sc->sc_disk.dk_label->d_partitions[DISKPART(dev)];
                return(0);

        /* do format operation, read or write */
        case DIOCRFORMAT:
        break;
        case DIOCWFORMAT:
        break;

        case DIOCSSTEP: /* set step rate */
        break;
        case DIOCSRETRIES: /* set # of retries */
        break;
        case DIOCKLABEL: /* keep/drop label on close? */
        break;
        case DIOCWLABEL: /* write en/disable label */
        break;

/*      case DIOCSBAD: / * set kernel dkbad */
        break; /* */
        case DIOCEJECT: /* eject removable disk */
        break;
        case ODIOCEJECT: /* eject removable disk */
        break;
        case DIOCLOCK: /* lock/unlock pack */
        break;

        /* get default label, clear label */
        case DIOCGDEFLABEL:
        break;
        case DIOCCLRLABEL:
        break;
        default:
                return(ENOTTY);
```

```
        }

        return(ENOTTY);
}
```

A rf.c

```
/*
 * Copyright (c) 2002 Jochen Kunz.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. The name of Jochen Kunz may not be used to endorse or promote
 *    products derived from this software without specific prior
 *    written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY JOCHEN KUNZ
 * ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 * TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL JOCHEN KUNZ
 * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

/*
TODO:
- Better LBN bound checking, block padding for SD disks.
- Formating / "Set Density"
- Better error handling / detaild error reason reportnig.
*/

/* autoconfig stuff */
#include <sys/param.h>
#include <sys/device.h>
#include <sys/conf.h>
#include "locators.h"
```

```
#include "ioconf.h"

/* bus_space / bus_dma */
#include <machine/bus.h>

/* UniBus / QBus specific stuff */
#include <dev/qbus/ubavar.h>

/* disk interface */
#include <sys/types.h>
#include <sys/disklabel.h>
#include <sys/disk.h>

/* general system data and functions */
#include <sys/systm.h>
#include <sys/ioctl.h>
#include <sys/ioccom.h>

/* physio / buffer handling */
#include <sys/buf.h>

/* tsleep / sleep / wakeup */
#include <sys/proc.h>
/* hz for above */
#include <sys/kernel.h>

/* bitdefinitions for RX211 */
#include <dev/qbus/rfreg.h>


#define RFS_DENS        0x0001          /* single or double density */
#define RFS_AD          0x0002          /* density auto detect */
#define RFS_NOTINIT     0x0000          /* not initialized */
#define RFS_PROBING     0x0010          /* density detect / verify started */
#define RFS_FBUF        0x0020          /* Fill Buffer */
#define RFS_EBUF        0x0030          /* Empty Buffer */
#define RFS_WSEC        0x0040          /* Write Sector */
#define RFS_RSEC        0x0050          /* Read Sector */
#define RFS_SMD         0x0060          /* Set Media Density */
#define RFS_RSTAT       0x0070          /* Read Status */
#define RFS_WDDS        0x0080          /* Write Deleted Data Sector */
#define RFS_REC         0x0090          /* Read Error Code */
#define RFS_IDLE        0x00a0          /* controller is idle */
```

```
#define RFS_CMDS        0x00f0          /* command mask */
#define RFS_OPEN_A      0x0100          /* partition a open */
#define RFS_OPEN_B      0x0200          /* partition b open */
#define RFS_OPEN_C      0x0400          /* partition c open */
#define RFS_OPEN_MASK   0x0f00          /* mask for open partitions */
#define RFS_OPEN_SHIFT  8               /* to shift 1 to get RFS_OPEN_A */
#define RFS_SETCMD(rf, state)   ((rf) = ((rf) & ~RFS_CMDS) | (state))



/* autoconfig stuff */
static int rfc_match(struct device *, struct cfdata *, void *);
static void rfc_attach(struct device *, struct device *, void *);
static int rf_match(struct device *, struct cfdata *, void *);
static void rf_attach(struct device *, struct device *, void *);
static int rf_print(void *, const char *);

/* device interfce functions / interface to disk(9) */
dev_type_open(rfopen);
dev_type_close(rfclose);
dev_type_read(rfread);
dev_type_write(rfwrite);
dev_type_ioctl(rfioctl);
dev_type_strategy(rfstrategy);
dev_type_dump(rfdump);
dev_type_size(rfsize);


/* Entries in block and character major device number switch table. */
const struct bdevsw rf_bdevsw = {
        rfopen,
        rfclose,
        rfstrategy,
        rfioctl,
        rfdump,
        rfsize,
        D_DISK
};

const struct cdevsw rf_cdevsw = {
        rfopen,
        rfclose,
        rfread,
```

```
	rfwrite,
	rfioctl,
	nostop,
	notty,
	nopoll,
	nommap,
	nokqfilter,
	D_DISK
};



struct rfc_softc {
	struct device sc_dev;		/* common device data */
	struct device *sc_childs[2];	/* child devices */
	struct evcnt sc_intr_count;	/* Interrupt counter for statistics */
	struct buf *sc_curbuf;		/* buf that is currently in work */
	bus_space_tag_t sc_iot;		/* bus_space IO tag */
	bus_space_handle_t sc_ioh;	/* bus_space IO handle */
	bus_dma_tag_t sc_dmat;		/* bus_dma DMA tag */
	bus_dmamap_t sc_dmam;		/* bus_dma DMA map */
	caddr_t sc_bufidx;		/* current position in buffer data */
	int sc_curchild;		/* child whose bufq is in work */
	int sc_bytesleft;		/* bytes left to transfer */
	u_int8_t type;			/* controller type, 1 or 2 */
};



CFATTACH_DECL(
	rfc,
	sizeof(struct rfc_softc),
	rfc_match,
	rfc_attach,
	NULL,
	NULL
);



struct rf_softc {
	struct device sc_dev;		/* common device data */
	struct disk sc_disk;		/* common disk device data */
```

```
	struct bufq_state sc_bufq;	/* queue of pending transfers */
	int sc_state;			/* state of drive */
	u_int8_t sc_dnum;		/* drive number, 0 or 1 */
};


CFATTACH_DECL(
	rf,
	sizeof(struct rf_softc),
	rf_match,
	rf_attach,
	NULL,
	NULL
);


struct rfc_attach_args {
	u_int8_t type;		/* controller type, 1 or 2 */
	u_int8_t dnum;		/* drive number, 0 or 1 */
};


struct dkdriver rfdkdriver = {
	rfstrategy
};


/* helper functions */
int rfc_sendcmd(struct rfc_softc *, int, int, int);
struct rf_softc* get_new_buf( struct rfc_softc *);
static void rfc_intr(void *);


/*
 * Issue a reset command to the controller and look for the bits in
 * RX2CS and RX2ES.
 * RX2CS_RX02 and / or RX2CS_DD can be set,
 * RX2ES has to be set, all other bits must be 0
```

```
 */
int
rfc_match(struct device *parent, struct cfdata *match, void *aux)
{
        struct uba_attach_args *ua = aux;
        int i;

        /* Issue reset command. */
        bus_space_write_2(ua->ua_iot, ua->ua_ioh, RX2CS, RX2CS_INIT);
        /* Wait for the controller to become ready, that is when
         * RX2CS_DONE, RX2ES_RDY and RX2ES_ID are set. */
        for (i = 0 ; i < 20 ; i++) {
                if ((bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2CS)
                    & RX2CS_DONE) != 0
                    && (bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2ES)
                    & (RX2ES_RDY | RX2ES_ID)) != 0)
                        break;
                DELAY(100000);  /* wait 100ms */
        }
        /*
         * Give up if the timeout has elapsed
         * and the controller is not ready.
         */
        if (i >= 20)
                return(0);
        /*
         * Issue a Read Status command with interrupt enabled.
         * The uba(4) driver wants to catch the interrupt to get the
         * interrupt vector and level of the device
         */
        bus_space_write_2(ua->ua_iot, ua->ua_ioh, RX2CS,
            RX2CS_RSTAT | RX2CS_IE);
        /*
         * Wait for command to finish, ignore errors and
         * abort if the controller does not respond within the timeout
         */
        for (i = 0 ; i < 20 ; i++) {
                if ((bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2CS)
                    & (RX2CS_DONE | RX2CS_IE)) != 0
                    && (bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2ES)
                    & RX2ES_RDY) != 0 )
                        return(1);
                DELAY(100000);  /* wait 100ms */
```

```
        }
        return(0);
}



/* #define RX02_PROBE 1 */
#ifdef RX02_PROBE
/*
 * Probe the density of an inserted floppy disk.
 * This is done by reading a sector from disk.
 * Return -1 on error, 0 on SD and 1 on DD.
 */
int rfcprobedens(struct rfc_softc *, int);
int
rfcprobedens(struct rfc_softc *rfc_sc, int dnum)
{
        int dens_flag;
        int i;

        dens_flag = 0;
        do {
                bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS,
                    RX2CS_RSEC | (dens_flag == 0 ? 0 : RX2CS_DD)
                    | (dnum == 0 ? 0 : RX2CS_US));
                /*
                 * Transfer request set?
                 * Wait 50us, the controller needs this time to setle
                 */
                DELAY(50);
                if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
                    & RX2CS_TR) == 0) {
                        printf("%s: did not respond to Read Sector CMD(1)\n",
                            rfc_sc->sc_dev.dv_xname);
                        return(-1);
                }
                bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2SA, 1);
                /* Wait 50us, the controller needs this time to setle */
                DELAY(50);
                if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
                    & RX2CS_TR) == 0) {
                        printf("%s: did not respond to Read Sector CMD(2)\n",
                            rfc_sc->sc_dev.dv_xname);
```

```
                        return(-1);
                }
                bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2TA, 1);
                /* Wait for the command to finish */
                for (i = 0 ; i < 200 ; i++) {
                        if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh,
                            RX2CS) & RX2CS_DONE) != 0)
                                break;
                        DELAY(10000);   /* wait 10ms */
                }
                if (i >= 200) {
                        printf("%s: did not respond to Read Sector CMD(3)\n",
                            rfc_sc->sc_dev.dv_xname);
                        return(-1);
                }
                if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
                    & RX2CS_ERR) == 0)
                        return(dens_flag);
        } while (rfc_sc->type == 2 && dens_flag++ == 0);
        return(-1);
}
#endif /* RX02_PROBE */



void
rfc_attach(struct device *parent, struct device *self, void *aux)
{
        struct rfc_softc *rfc_sc = (struct rfc_softc *)self;
        struct uba_attach_args *ua = aux;
        struct rfc_attach_args rfc_aa;
        int i;

        rfc_sc->sc_iot = ua->ua_iot;
        rfc_sc->sc_ioh = ua->ua_ioh;
        rfc_sc->sc_dmat = ua->ua_dmat;
        rfc_sc->sc_curbuf = NULL;
        /* Tell the QBus busdriver about our interrupt handler. */
        uba_intr_establish(ua->ua_icookie, ua->ua_cvec, rfc_intr, rfc_sc,
            &rfc_sc->sc_intr_count);
        /* Attach to the interrupt counter, see evcnt(9) */
        evcnt_attach_dynamic(&rfc_sc->sc_intr_count, EVCNT_TYPE_INTR,
            ua->ua_evcnt, rfc_sc->sc_dev.dv_xname, "intr");
```

```
	/* get a bus_dma(9) handle */
	i = bus_dmamap_create(rfc_sc->sc_dmat, RX2_BYTE_DD, 1, RX2_BYTE_DD, 0,
	    BUS_DMA_ALLOCNOW, &rfc_sc->sc_dmam);
	if (i != 0) {
		printf("rfc_attach: Error creating bus dma map: %d\n", i);
		return;
	}

	/* Issue reset command. */
	bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS, RX2CS_INIT);
	/*
	 * Wait for the controller to become ready, that is when
	 * RX2CS_DONE, RX2ES_RDY and RX2ES_ID are set.
	 */
	for (i = 0 ; i < 20 ; i++) {
		if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
		    & RX2CS_DONE) != 0
		    && (bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2ES)
		    & (RX2ES_RDY | RX2ES_ID)) != 0)
			break;
		DELAY(100000);	/* wait 100ms */
	}
	/*
	 * Give up if the timeout has elapsed
	 * and the controller is not ready.
	 */
	if (i >= 20) {
		printf(": did not respond to INIT CMD\n");
		return;
	}
	/* Is ths a RX01 or a RX02? */
	if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
	    & RX2CS_RX02) != 0) {
		rfc_sc->type = 2;
		rfc_aa.type = 2;
	} else {
		rfc_sc->type = 1;
		rfc_aa.type = 1;
	}
	printf(": RX0%d\n", rfc_sc->type);

#ifndef RX02_PROBE
	/*
```

```
	 * Bouth disk drievs and the controller are one physical unit.
	 * If we found the controller, there will be bouth disk drievs.
	 * So attach them.
	 */
	rfc_aa.dnum = 0;
	rfc_sc->sc_childs[0] = config_found(&rfc_sc->sc_dev, &rfc_aa,rf_print);
	rfc_aa.dnum = 1;
	rfc_sc->sc_childs[1] = config_found(&rfc_sc->sc_dev, &rfc_aa,rf_print);
#else /* RX02_PROBE */
	/*
	 * There are clones of the DEC RX system with standard shugart
	 * interface. In this case we can not be sure that there are
	 * bouth disk drievs. So we want to do a detection of attached
	 * drives. This is done by reading a sector from disk. This means
	 * that there must be a formated disk in the drive at boot time.
	 * This is bad, but I did not find an other way to detect the
	 * (non)existence of a floppy drive.
	 */
	if (rfcprobedens(rfc_sc, 0) >= 0) {
		rfc_aa.dnum = 0;
		rfc_sc->sc_childs[0] = config_found(&rfc_sc->sc_dev, &rfc_aa,
		    rf_print);
	} else
		rfc_sc->sc_childs[0] = NULL;
	if (rfcprobedens(rfc_sc, 1) >= 0) {
		rfc_aa.dnum = 1;
		rfc_sc->sc_childs[1] = config_found(&rfc_sc->sc_dev, &rfc_aa,
		    rf_print);
	} else
		rfc_sc->sc_childs[1] = NULL;
#endif /* RX02_PROBE */
	return;
}



int
rf_match(struct device *parent, struct cfdata *match, void *aux)
{
	struct rfc_attach_args *rfc_aa = aux;

	/*
	 * Only attach if the locator is wildcarded or
```

```
         * if the specified locator addresses the current device.
         */
        if (match->cf_loc[RFCCF_DRIVE] == RFCCF_DRIVE_DEFAULT ||
            match->cf_loc[RFCCF_DRIVE] == rfc_aa->dnum)
                return(1);
        return(0);
}



void
rf_attach(struct device *parent, struct device *self, void *aux)
{
        struct rf_softc *rf_sc = (struct rf_softc *)self;
        struct rfc_attach_args *rfc_aa = (struct rfc_attach_args *)aux;
        struct rfc_softc *rfc_sc;
        struct disklabel *dl;

        rfc_sc = (struct rfc_softc *)rf_sc->sc_dev.dv_parent;
        rf_sc->sc_dnum = rfc_aa->dnum;
        rf_sc->sc_state = 0;
        rf_sc->sc_disk.dk_name = rf_sc->sc_dev.dv_xname;
        rf_sc->sc_disk.dk_driver = &rfdkdriver;
        disk_attach(&rf_sc->sc_disk);
        dl = rf_sc->sc_disk.dk_label;
        dl->d_type = DTYPE_FLOPPY;              /* drive type */
        dl->d_magic = DISKMAGIC;                /* the magic number */
        dl->d_magic2 = DISKMAGIC;
        dl->d_typename[0] = 'R';
        dl->d_typename[1] = 'X';
        dl->d_typename[2] = '0';
        dl->d_typename[3] = rfc_sc->type == 1 ? '1' : '2';      /* type name */
        dl->d_typename[4] = '\0';
        dl->d_secsize = DEV_BSIZE;              /* bytes per sector */
        /*
         * Fill in some values to have a initialized data structure. Some
         * values will be reset by rfopen() depending on the actual density.
         */
        dl->d_nsectors = RX2_SECTORS;           /* sectors per track */
        dl->d_ntracks = 1;                                                              /* tracks per cylinder */
        dl->d_ncylinders = RX2_TRACKS;          /* cylinders per unit */
        dl->d_secpercyl = RX2_SECTORS;          /* sectors per cylinder */
        dl->d_secperunit = RX2_SECTORS * RX2_TRACKS;    /* sectors per unit */
```

```
        dl->d_rpm = 360;                        /* rotational speed */
        dl->d_interleave = 1;                   /* hardware sector interleave */
        /* number of partitions in following */
        dl->d_npartitions = MAXPARTITIONS;
        dl->d_bbsize = 0;               /* size of boot area at sn0, bytes */
        dl->d_sbsize = 0;               /* max size of fs superblock, bytes */
        /* number of sectors in partition */
        dl->d_partitions[0].p_size = 501;
        dl->d_partitions[0].p_offset = 0;       /* starting sector */
        dl->d_partitions[0].p_fsize = 0;        /* fs basic fragment size */
        dl->d_partitions[0].p_fstype = 0;       /* fs type */
        dl->d_partitions[0].p_frag = 0;         /* fs fragments per block */
        dl->d_partitions[1].p_size = RX2_SECTORS * RX2_TRACKS / 2;
        dl->d_partitions[1].p_offset = 0;       /* starting sector */
        dl->d_partitions[1].p_fsize = 0;        /* fs basic fragment size */
        dl->d_partitions[1].p_fstype = 0;       /* fs type */
        dl->d_partitions[1].p_frag = 0;         /* fs fragments per block */
        dl->d_partitions[2].p_size = RX2_SECTORS * RX2_TRACKS;
        dl->d_partitions[2].p_offset = 0;       /* starting sector */
        dl->d_partitions[2].p_fsize = 0;        /* fs basic fragment size */
        dl->d_partitions[2].p_fstype = 0;       /* fs type */
        dl->d_partitions[2].p_frag = 0;         /* fs fragments per block */
        bufq_alloc(&rf_sc->sc_bufq, BUFQ_DISKSORT | BUFQ_SORT_CYLINDER);
        printf("\n");
        return;
}



int
rf_print(void *aux, const char *name)
{
        struct rfc_attach_args *rfc_aa = aux;

        if (name != NULL)
                aprint_normal("RX0%d at %s", rfc_aa->type, name);
        aprint_normal(" drive %d", rfc_aa->dnum);
        return(UNCONF);
}



/* Send a command to the controller */
```

```
int
rfc_sendcmd(struct rfc_softc *rfc_sc, int cmd, int data1, int data2)
{

        /* Write command to CSR. */
        bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS, cmd);
        /* Wait 50us, the controller needs this time to setle. */
        DELAY(50);
        /* Write parameter 1 to DBR */
        if ((cmd & RX2CS_FC) != RX2CS_RSTAT) {
                /* Transfer request set? */
                if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
                    & RX2CS_TR) == 0) {
                        printf("%s: did not respond to CMD %x (1)\n",
                            rfc_sc->sc_dev.dv_xname, cmd);
                        return(-1);
                }
                bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2DB,
                    data1);
        }
        /* Write parameter 2 to DBR */
        if ((cmd & RX2CS_FC) <= RX2CS_RSEC || (cmd & RX2CS_FC) == RX2CS_WDDS) {
                /* Wait 50us, the controller needs this time to setle. */
                DELAY(50);
                /* Transfer request set? */
                if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
                    & RX2CS_TR) == 0) {
                        printf("%s: did not respond to CMD %x (2)\n",
                            rfc_sc->sc_dev.dv_xname, cmd);
                        return(-1);
                }
                bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2DB,
                    data2);
        }
        return(1);
}



void
rfstrategy(struct buf *buf)
{
        struct rf_softc *rf_sc;
```

```
	struct rfc_softc *rfc_sc;
	int i;

	i = DISKUNIT(buf->b_dev);
	if (i >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[i]) == NULL) {
		buf->b_flags |= B_ERROR;
		buf->b_error = ENXIO;
		biodone(buf);
		return;
	}
	rfc_sc = (struct rfc_softc *)rf_sc->sc_dev.dv_parent;
	/* We are going to operate on a non open dev? PANIC! */
	if ((rf_sc->sc_state & 1 << (DISKPART(buf->b_dev) + RFS_OPEN_SHIFT))
	    == 0)
		panic("rfstrategy: can not operate on non-open drive %s "
		    "partition %d", rf_sc->sc_dev.dv_xname,
		    DISKPART(buf->b_dev));
	if (buf->b_bcount == 0) {
		biodone(buf);
		return;
	}
	/*
	 * BUFQ_PUT() operates on b_rawblkno. rfstrategy() gets
	 * only b_blkno that is partition relative. As a floppy does not
	 * have partitions b_rawblkno == b_blkno.
	 */
	buf->b_rawblkno = buf->b_blkno;
	/*
	 * from sys/kern/subr_disk.c:
	 * Seek sort for disks.  We depend on the driver which calls us using
	 * b_resid as the current cylinder number.
	 */
	i = splbio();
	if (rfc_sc->sc_curbuf == NULL) {
		rfc_sc->sc_curchild = rf_sc->sc_dnum;
		rfc_sc->sc_curbuf = buf;
		rfc_sc->sc_bufidx = buf->b_un.b_addr;
		rfc_sc->sc_bytesleft = buf->b_bcount;
		rfc_intr(rfc_sc);
	} else {
		buf->b_resid = buf->b_blkno / RX2_SECTORS;
		BUFQ_PUT(&rf_sc->sc_bufq, buf);
		buf->b_resid = 0;
```

```
        }
        splx(i);
        return;
}



/*
 * Look if there is an other buffer in the bufferqueue of this drive
 * and start to process it if there is one.
 * If the bufferqueue is empty, look at the bufferqueue of the other drive
 * that is attached to this controller.
 * Start procesing the bufferqueue of the other drive if it isn't empty.
 * Return a pointer to the softc structure of the drive that is now
 * ready to process a buffer or NULL if there is no buffer in either queues.
 */
struct rf_softc*
get_new_buf( struct rfc_softc *rfc_sc)
{
        struct rf_softc *rf_sc;
        struct rf_softc *other_drive;

        rf_sc = (struct rf_softc *)rfc_sc->sc_childs[rfc_sc->sc_curchild];
        rfc_sc->sc_curbuf = BUFQ_GET(&rf_sc->sc_bufq);
        if (rfc_sc->sc_curbuf != NULL) {
                rfc_sc->sc_bufidx = rfc_sc->sc_curbuf->b_un.b_addr;
                rfc_sc->sc_bytesleft = rfc_sc->sc_curbuf->b_bcount;
        } else {
                RFS_SETCMD(rf_sc->sc_state, RFS_IDLE);
                other_drive = (struct rf_softc *)
                    rfc_sc->sc_childs[ rfc_sc->sc_curchild == 0 ? 1 : 0];
                if (other_drive != NULL
                    && BUFQ_PEEK(&other_drive->sc_bufq) != NULL) {
                        rfc_sc->sc_curchild = rfc_sc->sc_curchild == 0 ? 1 : 0;
                        rf_sc = other_drive;
                        rfc_sc->sc_curbuf = BUFQ_GET(&rf_sc->sc_bufq);
                        rfc_sc->sc_bufidx = rfc_sc->sc_curbuf->b_un.b_addr;
                        rfc_sc->sc_bytesleft = rfc_sc->sc_curbuf->b_bcount;
                } else
                        return(NULL);
        }
        return(rf_sc);
}
```

```
void
rfc_intr(void *intarg)
{
        struct rfc_softc *rfc_sc = intarg;
        struct rf_softc *rf_sc;
        int i;

        rf_sc = (struct rf_softc *)rfc_sc->sc_childs[rfc_sc->sc_curchild];
        do {
                /*
                 * First clean up from previous command...
                 */
                switch (rf_sc->sc_state & RFS_CMDS) {
                case RFS_PROBING:       /* density detect / verify started */
                        disk_unbusy(&rf_sc->sc_disk, 0, 1);
                        if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh,
                            RX2CS) & RX2CS_ERR) == 0) {
                                RFS_SETCMD(rf_sc->sc_state, RFS_IDLE);
                                wakeup(rf_sc);
                        } else {
                                if (rfc_sc->type == 2
                                    && (rf_sc->sc_state & RFS_DENS) == 0
                                    && (rf_sc->sc_state & RFS_AD) != 0) {
                                        /* retry at DD */
                                        rf_sc->sc_state |= RFS_DENS;
                                        disk_busy(&rf_sc->sc_disk);
                                        if (rfc_sendcmd(rfc_sc, RX2CS_RSEC
                                            | RX2CS_IE | RX2CS_DD |
                                            (rf_sc->sc_dnum == 0 ? 0 :
                                            RX2CS_US), 1, 1) < 0) {
                                                disk_unbusy(&rf_sc->sc_disk,
                                                    0, 1);
                                                RFS_SETCMD(rf_sc->sc_state,
                                                    RFS_NOTINIT);
                                                wakeup(rf_sc);
                                        }
                                } else {
                                        printf("%s: density error.\n",
                                            rf_sc->sc_dev.dv_xname);
                                        RFS_SETCMD(rf_sc->sc_state,RFS_NOTINIT);
```

```
                        wakeup(rf_sc);
                }
        }
        return;
case RFS_IDLE:  /* controller is idle */
        if (rfc_sc->sc_curbuf->b_bcount
            % ((rf_sc->sc_state & RFS_DENS) == 0
            ? RX2_BYTE_SD : RX2_BYTE_DD) != 0) {
                /*
                 * can only handle blocks that are a multiple
                 * of the physical block size
                 */
                rfc_sc->sc_curbuf->b_flags |= B_ERROR;
        }
        RFS_SETCMD(rf_sc->sc_state, (rfc_sc->sc_curbuf->b_flags
            & B_READ) != 0 ? RFS_RSEC : RFS_FBUF);
        break;
case RFS_RSEC:  /* Read Sector */
        disk_unbusy(&rf_sc->sc_disk, 0, 1);
        /* check for errors */
        if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh,
            RX2CS) & RX2CS_ERR) != 0) {
                /* should do more verbose error reporting */
                printf("rfc_intr: Error reading sector: %x\n",
                    bus_space_read_2(rfc_sc->sc_iot,
                    rfc_sc->sc_ioh, RX2ES) );
                rfc_sc->sc_curbuf->b_flags |= B_ERROR;
        }
        RFS_SETCMD(rf_sc->sc_state, RFS_EBUF);
        break;
case RFS_WSEC:  /* Write Sector */
        i = (rf_sc->sc_state & RFS_DENS) == 0
                ? RX2_BYTE_SD : RX2_BYTE_DD;
        disk_unbusy(&rf_sc->sc_disk, i, 0);
        /* check for errors */
        if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh,
            RX2CS) & RX2CS_ERR) != 0) {
                /* should do more verbose error reporting */
                printf("rfc_intr: Error writing sector: %x\n",
                    bus_space_read_2(rfc_sc->sc_iot,
                    rfc_sc->sc_ioh, RX2ES) );
                rfc_sc->sc_curbuf->b_flags |= B_ERROR;
                break;
```

```
        }
        if (rfc_sc->sc_bytesleft > i) {
                rfc_sc->sc_bytesleft -= i;
                rfc_sc->sc_bufidx += i;
        } else {
                biodone(rfc_sc->sc_curbuf);
                rf_sc = get_new_buf( rfc_sc);
                if (rf_sc == NULL)
                        return;
        }
        RFS_SETCMD(rf_sc->sc_state,
            (rfc_sc->sc_curbuf->b_flags & B_READ) != 0
            ? RFS_RSEC : RFS_FBUF);
        break;
case RFS_FBUF:  /* Fill Buffer */
        disk_unbusy(&rf_sc->sc_disk, 0, 0);
        bus_dmamap_unload(rfc_sc->sc_dmat, rfc_sc->sc_dmam);
        /* check for errors */
        if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh,
            RX2CS) & RX2CS_ERR) != 0) {
                /* should do more verbose error reporting */
                printf("rfc_intr: Error while DMA: %x\n",
                    bus_space_read_2(rfc_sc->sc_iot,
                    rfc_sc->sc_ioh, RX2ES));
                rfc_sc->sc_curbuf->b_flags |= B_ERROR;
        }
        RFS_SETCMD(rf_sc->sc_state, RFS_WSEC);
        break;
case RFS_EBUF:  /* Empty Buffer */
        i = (rf_sc->sc_state & RFS_DENS) == 0
            ? RX2_BYTE_SD : RX2_BYTE_DD;
        disk_unbusy(&rf_sc->sc_disk, i, 1);
        bus_dmamap_unload(rfc_sc->sc_dmat, rfc_sc->sc_dmam);
        /* check for errors */
        if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh,
            RX2CS) & RX2CS_ERR) != 0) {
                /* should do more verbose error reporting */
                printf("rfc_intr: Error while DMA: %x\n",
                    bus_space_read_2(rfc_sc->sc_iot,
                    rfc_sc->sc_ioh, RX2ES));
                rfc_sc->sc_curbuf->b_flags |= B_ERROR;
                break;
        }
```

```
        if (rfc_sc->sc_bytesleft > i) {
                rfc_sc->sc_bytesleft -= i;
                rfc_sc->sc_bufidx += i;
        } else {
                biodone(rfc_sc->sc_curbuf);
                rf_sc = get_new_buf( rfc_sc);
                if (rf_sc == NULL)
                        return;
        }
        RFS_SETCMD(rf_sc->sc_state,
            (rfc_sc->sc_curbuf->b_flags & B_READ) != 0
            ? RFS_RSEC : RFS_FBUF);
        break;
case RFS_NOTINIT: /* Device is not open */
case RFS_SMD:   /* Set Media Density */
case RFS_RSTAT: /* Read Status */
case RFS_WDDS:  /* Write Deleted Data Sector */
case RFS_REC:   /* Read Error Code */
default:
        panic("Impossible state in rfc_intr(1).\n");
}

if ((rfc_sc->sc_curbuf->b_flags & B_ERROR) != 0) {
        /*
         * An error occured while processing this buffer.
         * Finish it and try to get a new buffer to process.
         * Return if there are no buffers in the queues.
         * This loops until the queues are empty or a new
         * action was successfully scheduled.
         */
        rfc_sc->sc_curbuf->b_resid = rfc_sc->sc_bytesleft;
        rfc_sc->sc_curbuf->b_error = EIO;
        biodone(rfc_sc->sc_curbuf);
        rf_sc = get_new_buf( rfc_sc);
        if (rf_sc == NULL)
                return;
        continue;
}

/*
 * ... then initiate next command.
 */
switch (rf_sc->sc_state & RFS_CMDS) {
```

```
case RFS_EBUF:  /* Empty Buffer */
        i = bus_dmamap_load(rfc_sc->sc_dmat, rfc_sc->sc_dmam,
            rfc_sc->sc_bufidx, (rf_sc->sc_state & RFS_DENS) == 0
            ? RX2_BYTE_SD : RX2_BYTE_DD,
            rfc_sc->sc_curbuf->b_proc, BUS_DMA_NOWAIT);
        if (i != 0) {
                printf("rfc_intr: Error loading dmamap: %d\n",
                i);
                rfc_sc->sc_curbuf->b_flags |= B_ERROR;
                break;
        }
        disk_busy(&rf_sc->sc_disk);
        if (rfc_sendcmd(rfc_sc, RX2CS_EBUF | RX2CS_IE
            | ((rf_sc->sc_state & RFS_DENS) == 0 ? 0 : RX2CS_DD)
            | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
            | ((rfc_sc->sc_dmam->dm_segs[0].ds_addr
            & 0x30000) >>4), ((rf_sc->sc_state & RFS_DENS) == 0
            ? RX2_BYTE_SD : RX2_BYTE_DD) / 2,
            rfc_sc->sc_dmam->dm_segs[0].ds_addr & 0xffff) < 0) {
                disk_unbusy(&rf_sc->sc_disk, 0, 1);
                rfc_sc->sc_curbuf->b_flags |= B_ERROR;
                bus_dmamap_unload(rfc_sc->sc_dmat,
                rfc_sc->sc_dmam);
        }
        break;
case RFS_FBUF:  /* Fill Buffer */
        i = bus_dmamap_load(rfc_sc->sc_dmat, rfc_sc->sc_dmam,
            rfc_sc->sc_bufidx, (rf_sc->sc_state & RFS_DENS) == 0
            ? RX2_BYTE_SD : RX2_BYTE_DD,
            rfc_sc->sc_curbuf->b_proc, BUS_DMA_NOWAIT);
        if (i != 0) {
                printf("rfc_intr: Error loading dmamap: %d\n",
                    i);
                rfc_sc->sc_curbuf->b_flags |= B_ERROR;
                break;
        }
        disk_busy(&rf_sc->sc_disk);
        if (rfc_sendcmd(rfc_sc, RX2CS_FBUF | RX2CS_IE
            | ((rf_sc->sc_state & RFS_DENS) == 0 ? 0 : RX2CS_DD)
            | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
            | ((rfc_sc->sc_dmam->dm_segs[0].ds_addr
            & 0x30000)>>4), ((rf_sc->sc_state & RFS_DENS) == 0
            ? RX2_BYTE_SD : RX2_BYTE_DD) / 2,
```

```
                rfc_sc->sc_dmam->dm_segs[0].ds_addr & 0xffff) < 0) {
                    disk_unbusy(&rf_sc->sc_disk, 0, 0);
                    rfc_sc->sc_curbuf->b_flags |= B_ERROR;
                    bus_dmamap_unload(rfc_sc->sc_dmat,
                        rfc_sc->sc_dmam);
            }
            break;
    case RFS_WSEC:  /* Write Sector */
            i = (rfc_sc->sc_curbuf->b_bcount - rfc_sc->sc_bytesleft
                + rfc_sc->sc_curbuf->b_blkno * DEV_BSIZE) /
                ((rf_sc->sc_state & RFS_DENS) == 0
                ? RX2_BYTE_SD : RX2_BYTE_DD);
            if (i > RX2_TRACKS * RX2_SECTORS) {
                    rfc_sc->sc_curbuf->b_flags |= B_ERROR;
                    break;
            }
            disk_busy(&rf_sc->sc_disk);
            if (rfc_sendcmd(rfc_sc, RX2CS_WSEC | RX2CS_IE
                | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
                | ((rf_sc->sc_state& RFS_DENS) == 0 ? 0 : RX2CS_DD),
                i % RX2_SECTORS + 1, i / RX2_SECTORS) < 0) {
                    disk_unbusy(&rf_sc->sc_disk, 0, 0);
                    rfc_sc->sc_curbuf->b_flags |= B_ERROR;
            }
            break;
    case RFS_RSEC:  /* Read Sector */
            i = (rfc_sc->sc_curbuf->b_bcount - rfc_sc->sc_bytesleft
                + rfc_sc->sc_curbuf->b_blkno * DEV_BSIZE) /
                ((rf_sc->sc_state & RFS_DENS) == 0
                ? RX2_BYTE_SD : RX2_BYTE_DD);
            if (i > RX2_TRACKS * RX2_SECTORS) {
                    rfc_sc->sc_curbuf->b_flags |= B_ERROR;
                    break;
            }
            disk_busy(&rf_sc->sc_disk);
            if (rfc_sendcmd(rfc_sc, RX2CS_RSEC | RX2CS_IE
                | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
                | ((rf_sc->sc_state& RFS_DENS) == 0 ? 0 : RX2CS_DD),
                i % RX2_SECTORS + 1, i / RX2_SECTORS) < 0) {
                    disk_unbusy(&rf_sc->sc_disk, 0, 1);
                    rfc_sc->sc_curbuf->b_flags |= B_ERROR;
            }
            break;
```

```
                case RFS_NOTINIT: /* Device is not open */
                case RFS_PROBING: /* density detect / verify started */
                case RFS_IDLE:  /* controller is idle */
                case RFS_SMD:   /* Set Media Density */
                case RFS_RSTAT: /* Read Status */
                case RFS_WDDS:  /* Write Deleted Data Sector */
                case RFS_REC:   /* Read Error Code */
                default:
                        panic("Impossible state in rfc_intr(2).\n");
                }

                if ((rfc_sc->sc_curbuf->b_flags & B_ERROR) != 0) {
                        /*
                         * An error occured while processing this buffer.
                         * Finish it and try to get a new buffer to process.
                         * Return if there are no buffers in the queues.
                         * This loops until the queues are empty or a new
                         * action was successfully scheduled.
                         */
                        rfc_sc->sc_curbuf->b_resid = rfc_sc->sc_bytesleft;
                        rfc_sc->sc_curbuf->b_error = EIO;
                        biodone(rfc_sc->sc_curbuf);
                        rf_sc = get_new_buf( rfc_sc);
                        if (rf_sc == NULL)
                                return;
                        continue;
                }
        } while ( 1 == 0 /* CONSTCOND */ );
        return;
}



int
rfdump(dev_t dev, daddr_t blkno, caddr_t va, size_t size)
{

        /* A 0.5MB floppy is much to small to take a system dump... */
        return(ENXIO);
}
```

```
int
rfsize(dev_t dev)
{

        return(-1);
}



int
rfopen(dev_t dev, int oflags, int devtype, struct proc *p)
{
        struct rf_softc *rf_sc;
        struct rfc_softc *rfc_sc;
        struct disklabel *dl;
        int unit;

        unit = DISKUNIT(dev);
        if (unit >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[unit]) == NULL) {
                return(ENXIO);
        }
        rfc_sc = (struct rfc_softc *)rf_sc->sc_dev.dv_parent;
        dl = rf_sc->sc_disk.dk_label;
        switch (DISKPART(dev)) {
                case 0:                 /* Part. a is single density. */
                        /* opening in single and double density is sensless */
                        if ((rf_sc->sc_state & RFS_OPEN_B) != 0 )
                                return(ENXIO);
                        rf_sc->sc_state &= ~RFS_DENS;
                        rf_sc->sc_state &= ~RFS_AD;
                        rf_sc->sc_state |= RFS_OPEN_A;
                break;
                case 1:                 /* Part. b is double density. */
                        /*
                         * Opening a singe density only drive in double
                         * density or simultaneous opening in single and
                         * double density is sensless.
                         */
                        if (rfc_sc->type == 1
                            || (rf_sc->sc_state & RFS_OPEN_A) != 0 )
                                return(ENXIO);
                        rf_sc->sc_state |= RFS_DENS;
                        rf_sc->sc_state &= ~RFS_AD;
```

```
                        rf_sc->sc_state |= RFS_OPEN_B;
                break;
                case 2:                 /* Part. c is auto density. */
                        rf_sc->sc_state |= RFS_AD;
                        rf_sc->sc_state |= RFS_OPEN_C;
                break;
                default:
                        return(ENXIO);
                break;
        }
        if ((rf_sc->sc_state & RFS_CMDS) == RFS_NOTINIT) {
                rfc_sc->sc_curchild = rf_sc->sc_dnum;
                /*
                 * Controller is idle and density is not detected.
                 * Start a density probe by issuing a read sector command
                 * and sleep until the density probe finished.
                 * Due to this it is impossible to open unformated media.
                 * As the RX02/02 is not able to format its own media,
                 * media must be purchased preformated. fsck DEC marketing!
                 */
                RFS_SETCMD(rf_sc->sc_state, RFS_PROBING);
                disk_busy(&rf_sc->sc_disk);
                if (rfc_sendcmd(rfc_sc, RX2CS_RSEC | RX2CS_IE
                    | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
                    | ((rf_sc->sc_state & RFS_DENS) == 0 ? 0 : RX2CS_DD),
                    1, 1) < 0) {
                        rf_sc->sc_state = 0;
                        return(ENXIO);
                }
                /* wait max. 2 sec for density probe to finish */
                if (tsleep(rf_sc, PRIBIO | PCATCH, "density probe", 2 * hz)
                    != 0 || (rf_sc->sc_state & RFS_CMDS) == RFS_NOTINIT) {
                        /* timeout elapsed and / or somthing went wrong */
                        rf_sc->sc_state = 0;
                        return(ENXIO);
                }
        }
        /* disklabel. We use different fake geometries for SD and DD. */
        if ((rf_sc->sc_state & RFS_DENS) == 0) {
                dl->d_nsectors = 10;            /* sectors per track */
                dl->d_secpercyl = 10;           /* sectors per cylinder */
                dl->d_ncylinders = 50;          /* cylinders per unit */
                dl->d_secperunit = 501; /* sectors per unit */
```

```
                /* number of sectors in partition */
                dl->d_partitions[2].p_size = 500;
        } else {
                dl->d_nsectors = RX2_SECTORS / 2;  /* sectors per track */
                dl->d_secpercyl = RX2_SECTORS / 2; /* sectors per cylinder */
                dl->d_ncylinders = RX2_TRACKS;     /* cylinders per unit */
                /* sectors per unit */
                dl->d_secperunit = RX2_SECTORS * RX2_TRACKS / 2;
                /* number of sectors in partition */
                dl->d_partitions[2].p_size = RX2_SECTORS * RX2_TRACKS / 2;
        }
        return(0);
}



int
rfclose(dev_t dev, int fflag, int devtype, struct proc *p)
{
        struct rf_softc *rf_sc;
        int unit;

        unit = DISKUNIT(dev);
        if (unit >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[unit]) == NULL) {
                return(ENXIO);
        }
        if ((rf_sc->sc_state & 1 << (DISKPART(dev) + RFS_OPEN_SHIFT)) == 0)
                panic("rfclose: can not close on non-open drive %s "
                    "partition %d", rf_sc->sc_dev.dv_xname, DISKPART(dev));
        else
                rf_sc->sc_state &= ~(1 << (DISKPART(dev) + RFS_OPEN_SHIFT));
        if ((rf_sc->sc_state & RFS_OPEN_MASK) == 0)
                rf_sc->sc_state = 0;
        return(0);
}



int
rfread(dev_t dev, struct uio *uio, int ioflag)
{

        return(physio(rfstrategy, NULL, dev, B_READ, minphys, uio));
```

```
}



int
rfwrite(dev_t dev, struct uio *uio, int ioflag)
{

        return(physio(rfstrategy, NULL, dev, B_WRITE, minphys, uio));
}



int
rfioctl(dev_t dev, u_long cmd, caddr_t data, int fflag, struct proc *p)
{
        struct rf_softc *rf_sc;
        int unit;

        unit = DISKUNIT(dev);
        if (unit >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[unit]) == NULL) {
                return(ENXIO);
        }
        /* We are going to operate on a non open dev? PANIC! */
        if ((rf_sc->sc_state & 1 << (DISKPART(dev) + RFS_OPEN_SHIFT)) == 0)
                panic("rfioctl: can not operate on non-open drive %s "
                    "partition %d", rf_sc->sc_dev.dv_xname, DISKPART(dev));
        switch (cmd) {
        /* get and set disklabel; DIOCGPART used internally */
        case DIOCGDINFO: /* get */
                memcpy(data, rf_sc->sc_disk.dk_label,
                    sizeof(struct disklabel));
                return(0);
        case DIOCSDINFO: /* set */
                return(0);
        case DIOCWDINFO: /* set, update disk */
                return(0);
        case DIOCGPART:  /* get partition */
                ((struct partinfo *)data)->disklab = rf_sc->sc_disk.dk_label;
                ((struct partinfo *)data)->part =
                    &rf_sc->sc_disk.dk_label->d_partitions[DISKPART(dev)];
                return(0);
```

```
        /* do format operation, read or write */
        case DIOCRFORMAT:
        break;
        case DIOCWFORMAT:
        break;

        case DIOCSSTEP: /* set step rate */
        break;
        case DIOCSRETRIES: /* set # of retries */
        break;
        case DIOCKLABEL: /* keep/drop label on close? */
        break;
        case DIOCWLABEL: /* write en/disable label */
        break;

/*      case DIOCSBAD: / * set kernel dkbad */
        break; /* */
        case DIOCEJECT: /* eject removable disk */
        break;
        case ODIOCEJECT: /* eject removable disk */
        break;
        case DIOCLOCK: /* lock/unlock pack */
        break;

        /* get default label, clear label */
        case DIOCGDEFLABEL:
        break;
        case DIOCCLRLABEL:
        break;
        default:
                return(ENOTTY);
        }

        return(ENOTTY);
}
```

B rfreg.h

```
/*
 * Copyright (c) 2002 Jochen Kunz.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. The name of Jochen Kunz may not be used to endorse or promote
 *    products derived from this software without specific prior
 *    written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY JOCHEN KUNZ
 * ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 * TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL JOCHEN KUNZ
 * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */



/* Registers in Uni/QBus IO space. */
#define RX2CS   0       /* Command and Status Register */
#define RX2DB   2       /* Data Buffer Register */
/* RX2DB is depending on context: */
#define RX2BA   2       /* Bus Address Register */
#define RX2TA   2       /* Track Address Register */
#define RX2SA   2       /* Sector Address Register */
#define RX2WC   2       /* Word Count Register */
#define RX2ES   2       /* Error and Status Register */
```

```
/* Bitdefinitions of CSR. */
#define RX2CS_ERR       0x8000  /* Error                                RO */
#define RX2CS_INIT      0x4000  /* Initialize                           WO */
#define RX2CS_UAEBH     0x2000  /* Unibus address extension high bit    WO */
#define RX2CS_UAEBI     0x1000  /* Unibus address extension low bit     WO */
#define RX2CS_RX02      0x0800  /* RX02                                 RO */
/*                      0x0400     Not Used                             -- */
/*                      0x0200     Not Used                             -- */
#define RX2CS_DD        0x0100  /* Double Density                       R/W */
#define RX2CS_TR        0x0080  /* Transfer Request                     RO */
#define RX2CS_IE        0x0040  /* Interrupt Enable                     R/W */
#define RX2CS_DONE      0x0020  /* Done                                 RO */
#define RX2CS_US        0x0010  /* Unit Select                          WO */
#define RX2CS_FCH       0x0008  /* Function Code high bit               WO */
#define RX2CS_FCM       0x0004  /* Function Code mid bit                WO */
#define RX2CS_FCL       0x0002  /* Function Code low bit                WO */
#define RX2CS_GO        0x0001  /* Go                                   WO */
#define RX2CS_NU        0x0600  /* not used bits                        -- */


#define RX2CS_UAEB      ( RX2CS_UAEBH | RX2CS_UAEBI )
#define RX2CS_FC        ( RX2CS_FCH | RX2CS_FCM | RX2CS_FCL )


/* Commands of the controller and parameter cont. */
#define RX2CS_FBUF      001     /* Fill Buffer, word count and bus address */
#define RX2CS_EBUF      003     /* Empty Buffer, word count and bus address */
#define RX2CS_WSEC      005     /* Write Sector, sector and track */
#define RX2CS_RSEC      007     /* Read Sector, sector and track */
#define RX2CS_SMD       011     /* Set Media Density, ??? */
#define RX2CS_RSTAT     013     /* Read Status, no params */
#define RX2CS_WDDS      015     /* Write Deleted Data Sector, sector and track */
#define RX2CS_REC       017     /* Read Error Code, bus address */


/* Track Address Register */
#define RX2TA_MASK      0x7f


/* Sector Address Register */
#define RX2SA_MASK      0x1f
```

```
/* Word Count Register */
#define RX2WC_MASK      0x7f


/* Bitdefinitions of RX2ES. */
/*                      <15-12> Not Used                -- */
#define RX2ES_NEM       0x0800  /* Non-Existend Memory  RO */
#define RX2ES_WCO       0x0400  /* Word Count Overflow  RO */
/*                      0x0200     Not Used             RO */
#define RX2ES_US        0x0010  /* Unit Select          RO */
#define RX2ES_RDY       0x0080  /* Ready                RO */
#define RX2ES_DEL       0x0040  /* Deleted Data         RO */
#define RX2ES_DD        0x0020  /* Double Density       RO */
#define RX2ES_DE        0x0010  /* Density Error        RO */
#define RX2ES_ACL       0x0008  /* AC Lost              RO */
#define RX2ES_ID        0x0004  /* Initialize Done      RO */
/*                      0x0002     Not Used             -- */
#define RX2ES_CRCE      0x0001  /* CRC Error            RO */
#define RX2ES_NU        0xF202  /* not used bits        -- */


#define RX2_TRACKS      77      /* number of tracks */
#define RX2_SECTORS     26      /* number of sectors / track */
#define RX2_BYTE_SD     128     /* number of bytes / sector in single density */
#define RX2_BYTE_DD     256     /* number of bytes / sector in double density */
#define RX2_HEADS       1       /* number of heads */
```

C Lizenz

D Versionsgeschichte

1.0 Erste öffentliche Ausgabe.

1.0.1 Korrektur diverser Tippfehler.

E Bibliographie

Literatur

[McK 99] Twenty Years of Berkeley Unix
From AT&T-Owned to Freely Redistributable
http://www.oreilly.com/catalog/opensources/book/kirkmck.html

[Tho] A Machine-Independent DMA Framework for NetBSD
Jason Thorpe
http://www.de.netbsd.org/Documentation/kernel/bus_dma.ps

[Li 77] A Commentary on the Sixth Edition UNIX Operating System
J. Lions, Department of Computer Science, The University of New South Wales

Index